External Sorting
Implementing Relational Operators

Readings

- [RG] Ch. 13 (sorting)

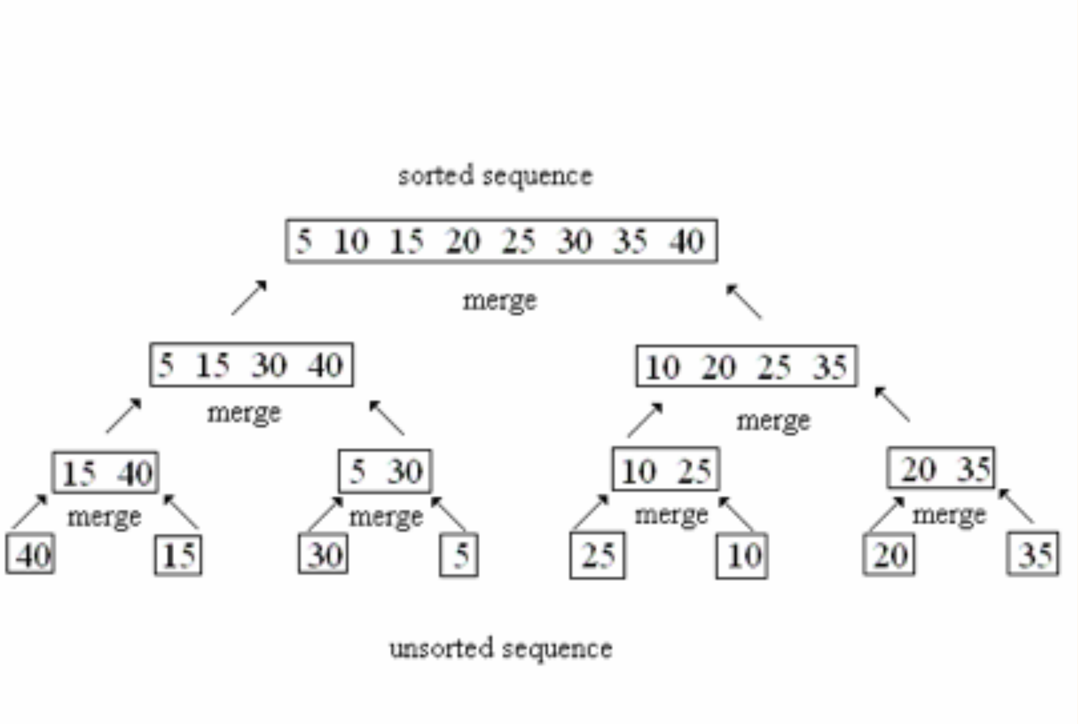
Where we are

- Working our way up from hardware
  - Disks
  - File abstraction that supports insert/delete/scan
  - Indexing for efficient access

External sorting

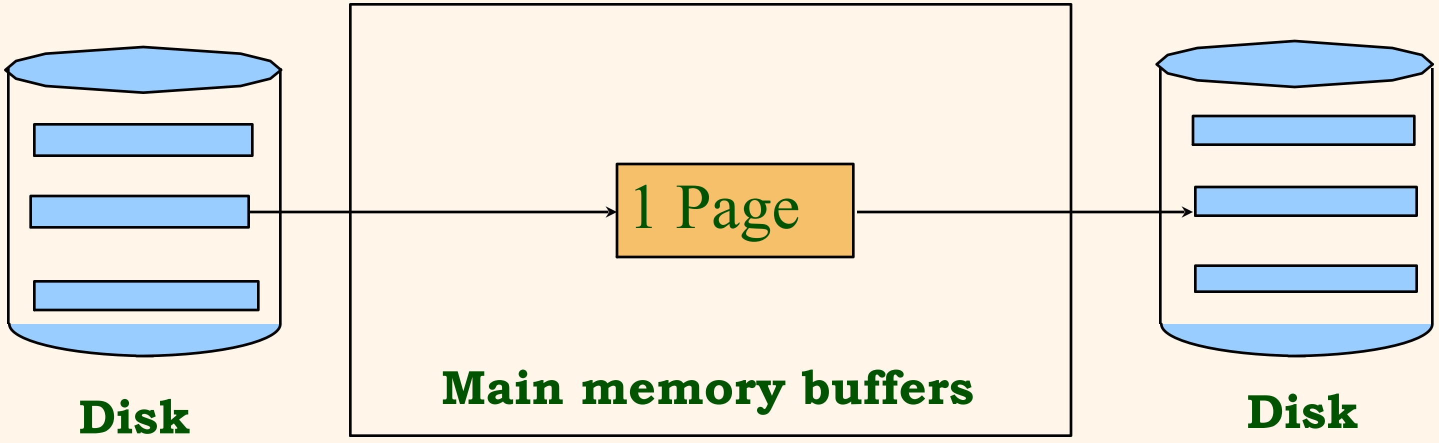
- Sorting a large amount of data that does not fit in RAM
  - E.g., sort 1TB of data with 16G of RAM
- Why sorting?
  - Useful building block in a variety of DB queries
    - Eliminate duplicates in a relation
    - Some join implementations
    - Or may just want data in sorted order
- Benchmarks and competitions
  - See http://sortbenchmark.org/

Remember merge sort?

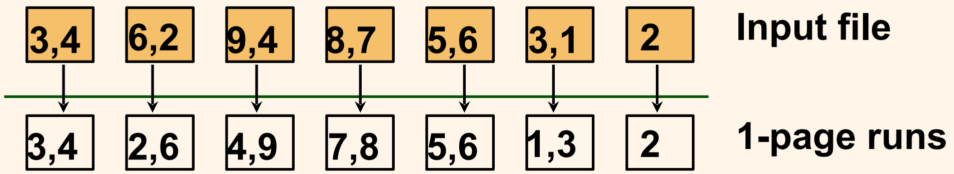


2-Way External Merge Sort

- Pass 0: Read a page at a time, sort it using your favorite algorithm, write it
  - Only one buffer page used (could use more if we have more)



Two-Way External Merge Sort: Pass 0



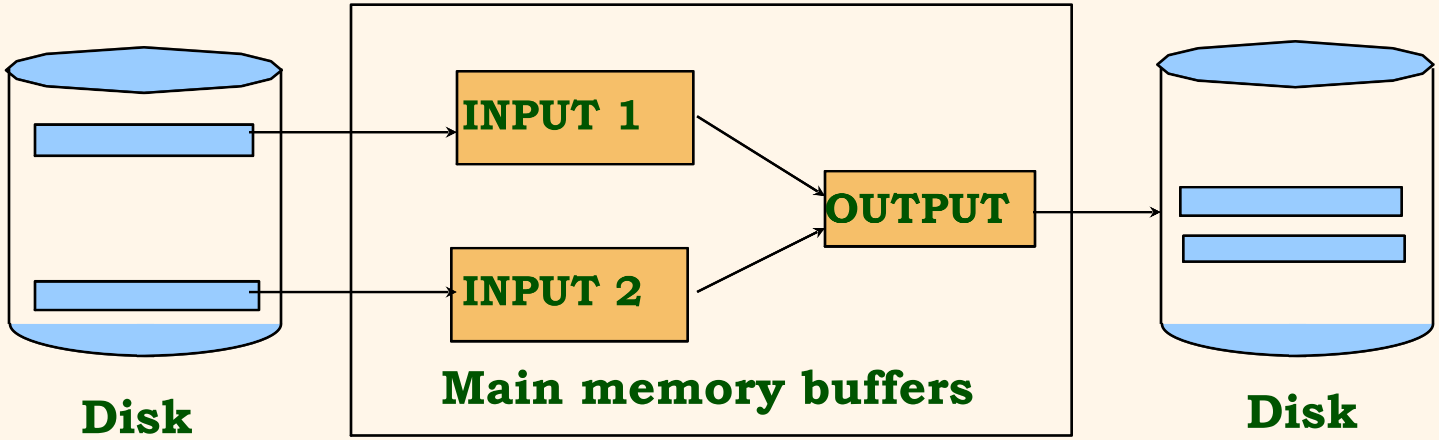
- Assume input file with N data pages
- What is the cost of Pass 0 (in terms of # I/Os)?

Definition:

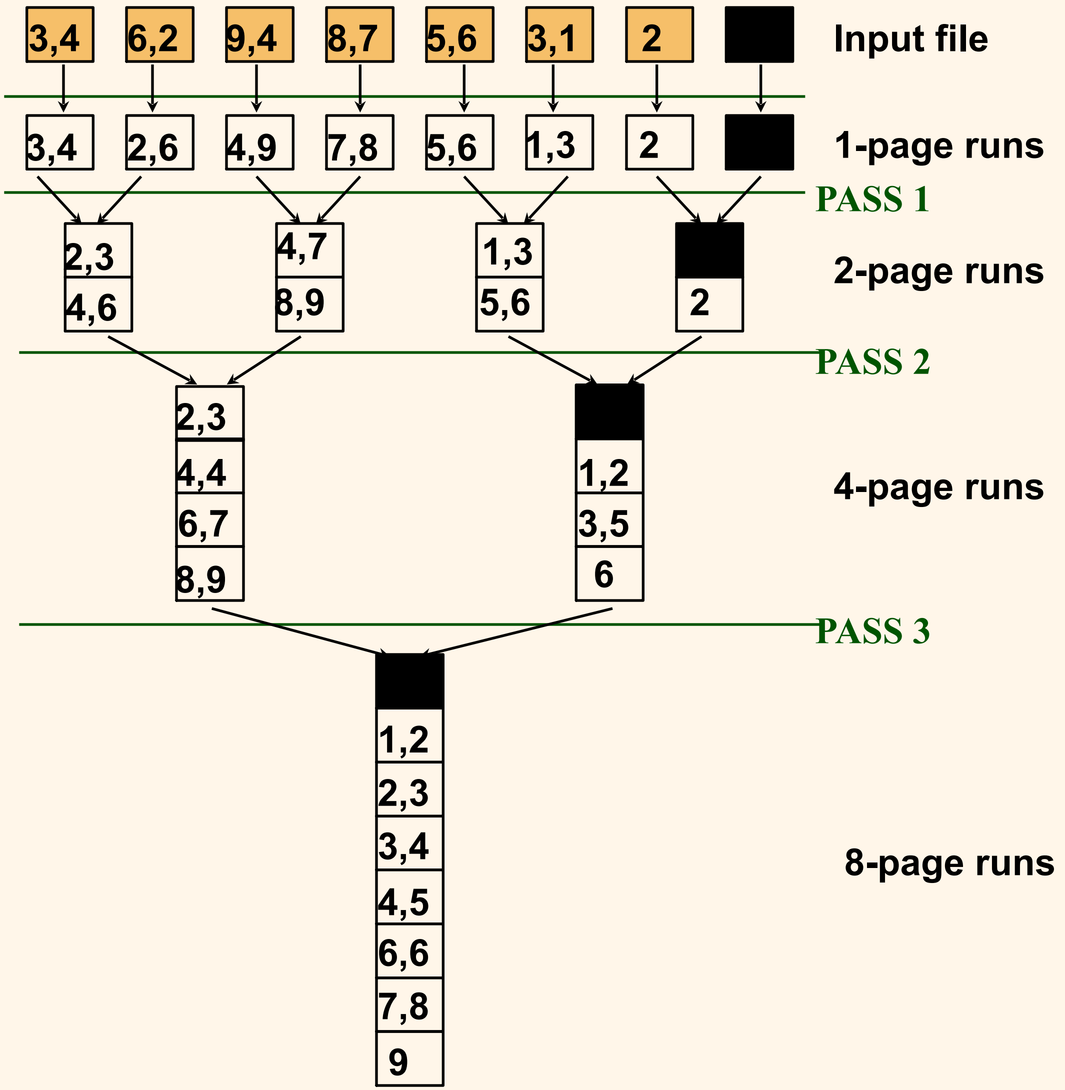
- A *run* is a sorted portion of the file
- Pass 0 generates some number of *runs* (how many??)

2-Way External Merge Sort

- Now: make more passes to merge runs
  - Pass 1: Merge two runs of length 1 (page)
  - Pass 2: Merge two runs of length 2 (pages)
  - … until 1 run of length N
  - Three buffer pages used



2-Way External Merge Sort



2-Way External Merge Sort: Analysis

- Total I/O cost for sorting file with N pages

- Cost of Pass 0 = $2N$

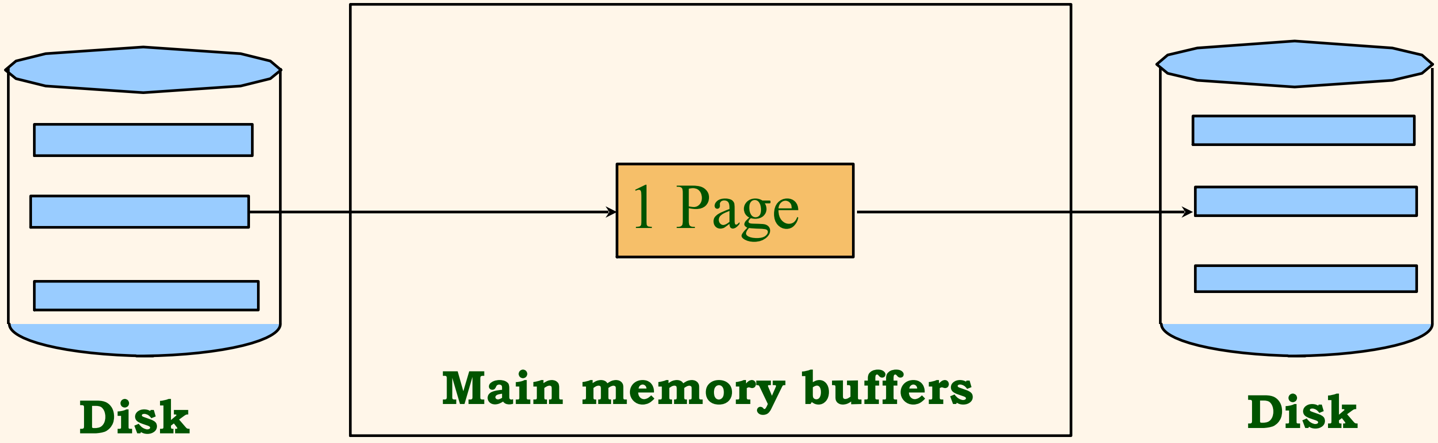
  - Number of merge passes = $\lceil \log_2 N \rceil$
  - Cost of each merge pass = $2N$
  - Cost of all merge passes = $2N \times \lceil \log_2 N \rceil$
  - Total cost = $2N\left(\lceil \log_2 N \rceil + 1\right)$

General External Merge Sort

- General case where B buffer pages are available
- Let's see how the calculations change

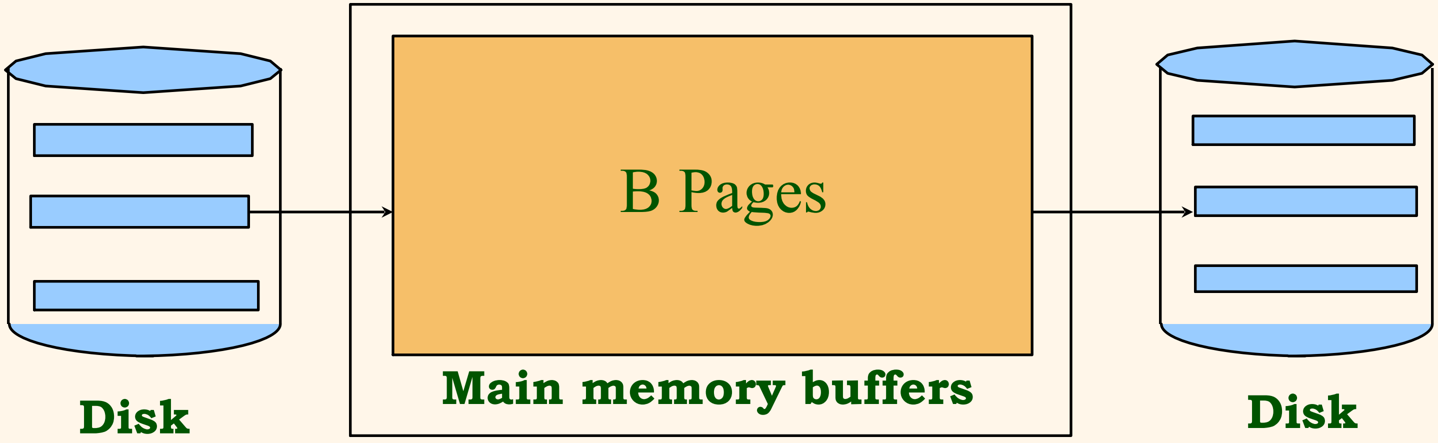
2-Way External Merge Sort

- Pass 0: read a page at a time, sort it using your favorite algorithm, write it
  - Only one buffer page used
- How can this be modified if B buffer pages are available?



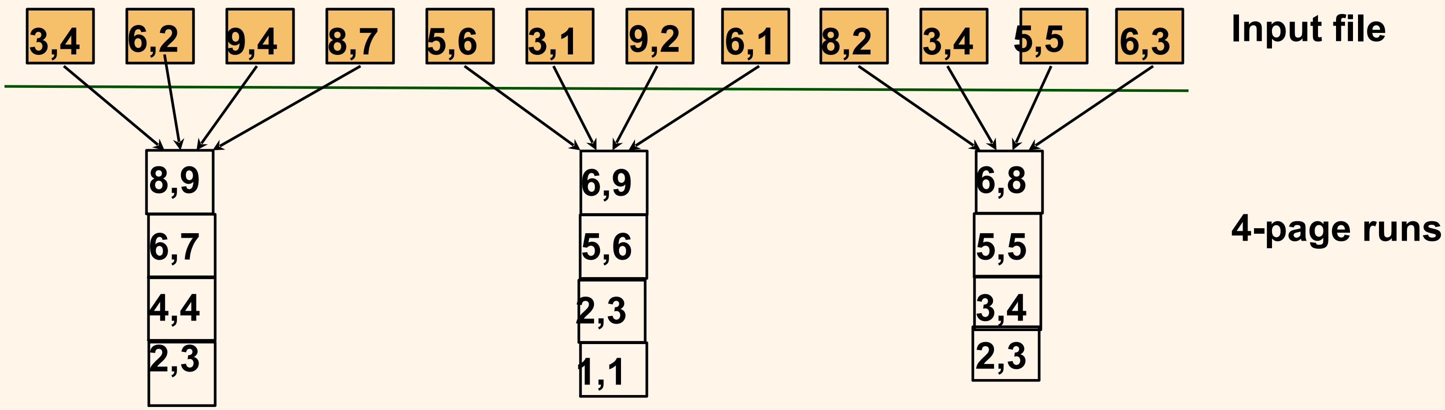
General External Merge Sort

- Read B pages at a time, sort B pages in main memory, and write out B pages
- Length of each run = B pages
- Assuming N input pages, number of runs = N/B
- Cost = $2N$



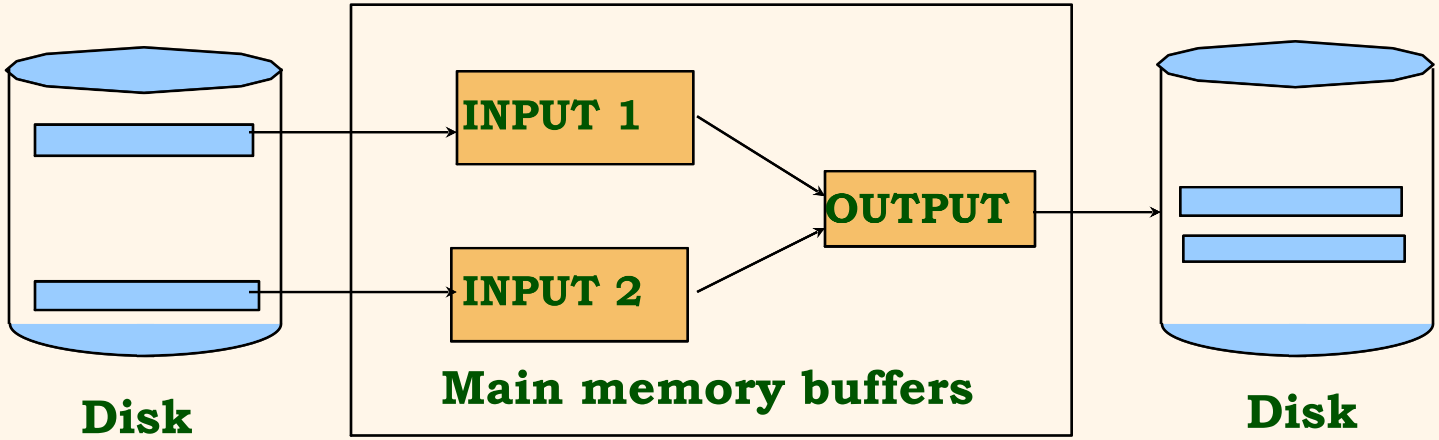
General External Merge Sort: Pass 0

- # buffer pages B = 4



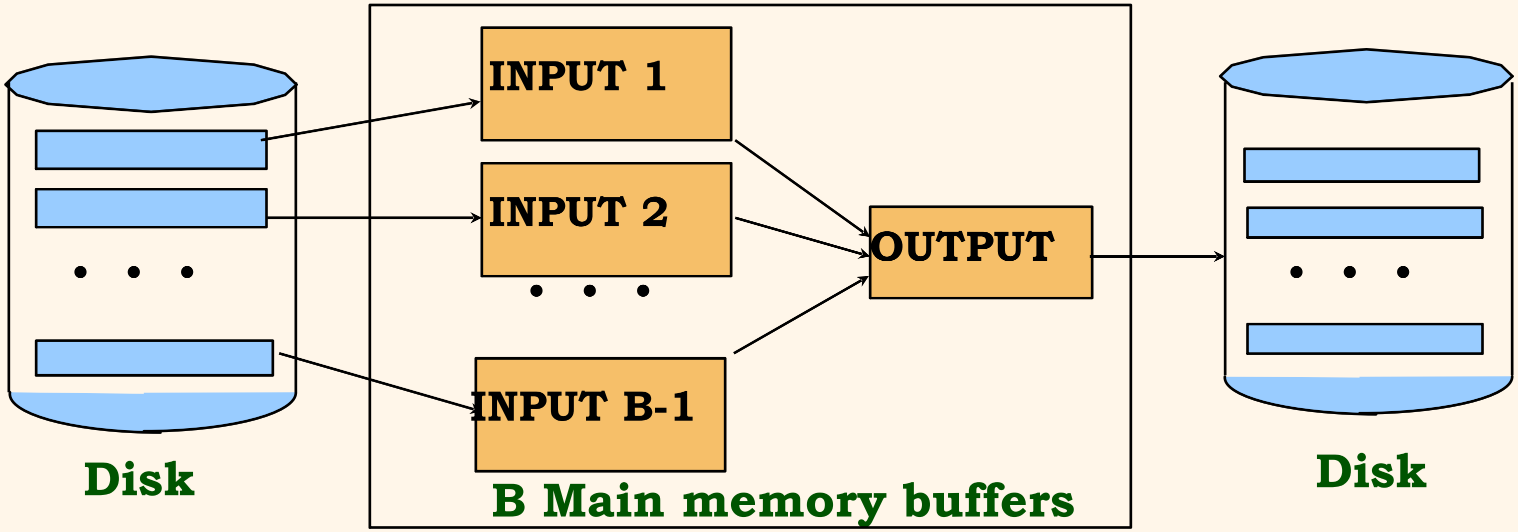
2-Way External Merge Sort

- Merge passes: Make multiple passes to merge runs
  - Pass 1: Merge two runs of length 1 (page)
  - Pass 2: Merge two runs of length 2 (pages)
  - … until 1 run of length N
  - Three buffer pages used
- How can this be modified if B buffer pages available?



General External Merge Sort

- Make multiple passes to merge runs
  - Pass 1: Produce runs of length B(B-1) pages
  - Pass 2: Produce runs of length $B(B-1)^2$ pages
  - …
  - Pass P: Produce runs of length $B(B-1)^P$ pages

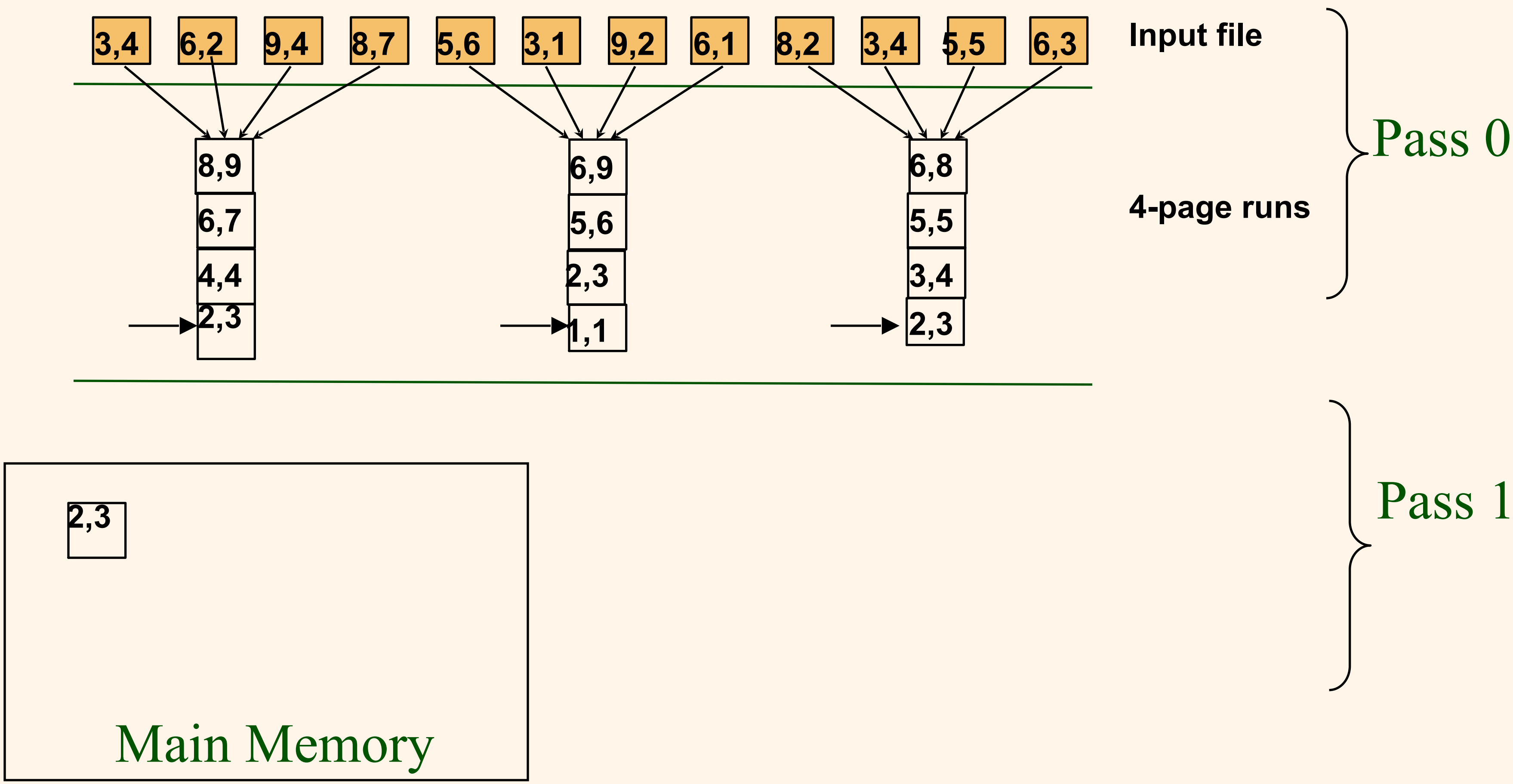


Definition

- Merge *fan-in* – number of runs being merged in each merge pass
  - What is the value of this fan-in?

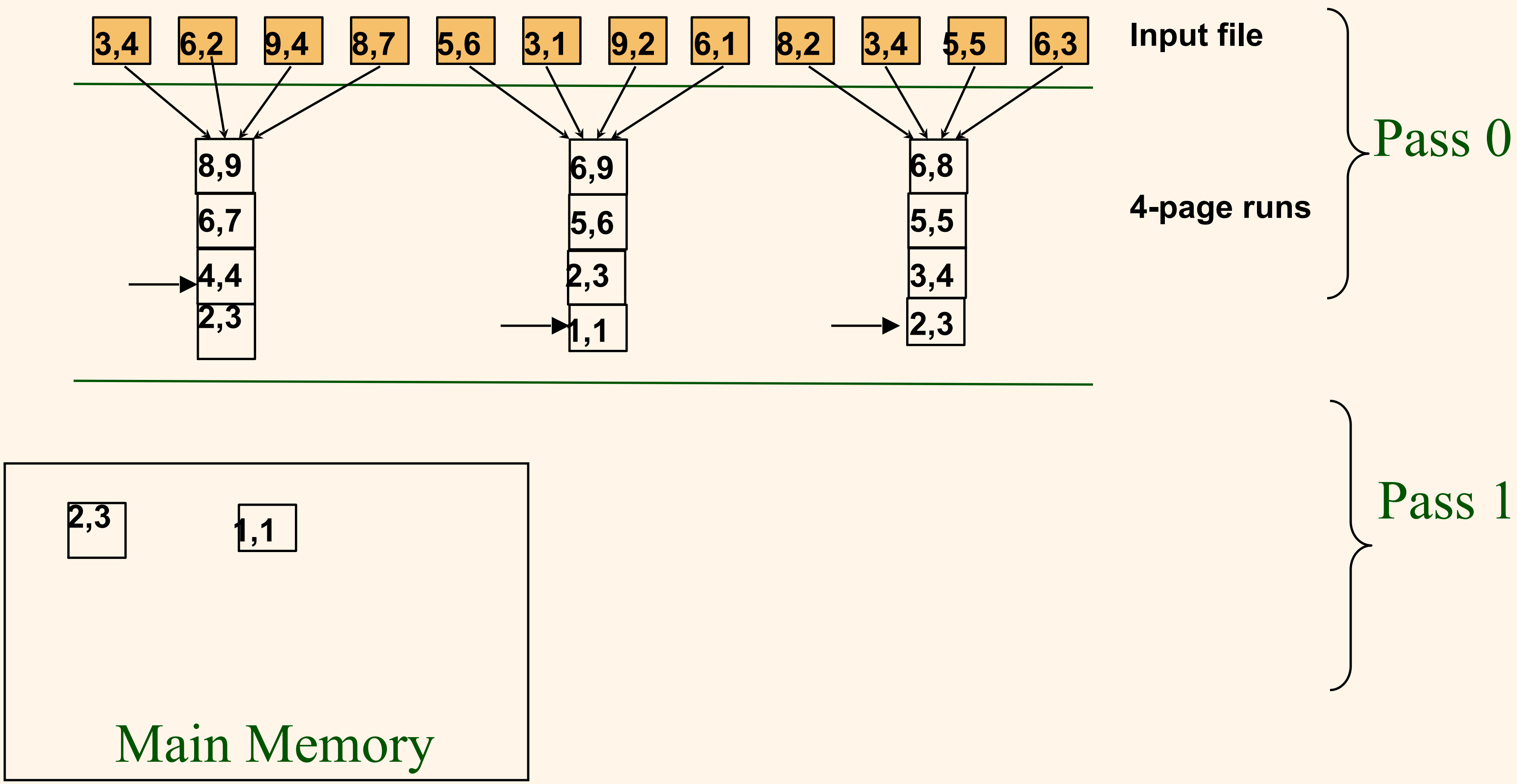
General External Merge Sort: Merge

- # buffer pages B = 4



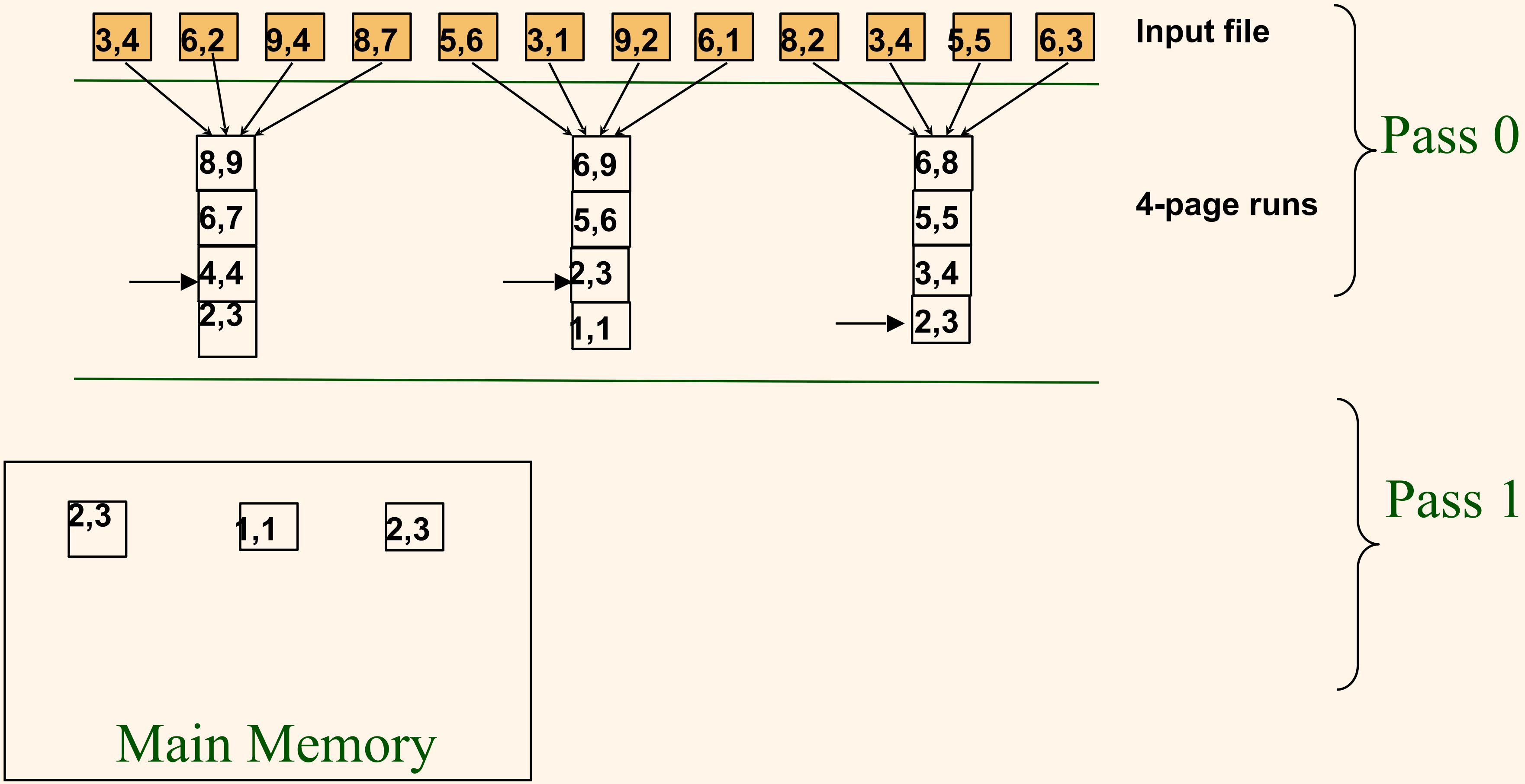
General External Merge Sort: Phase 2

- # buffer pages B = 4



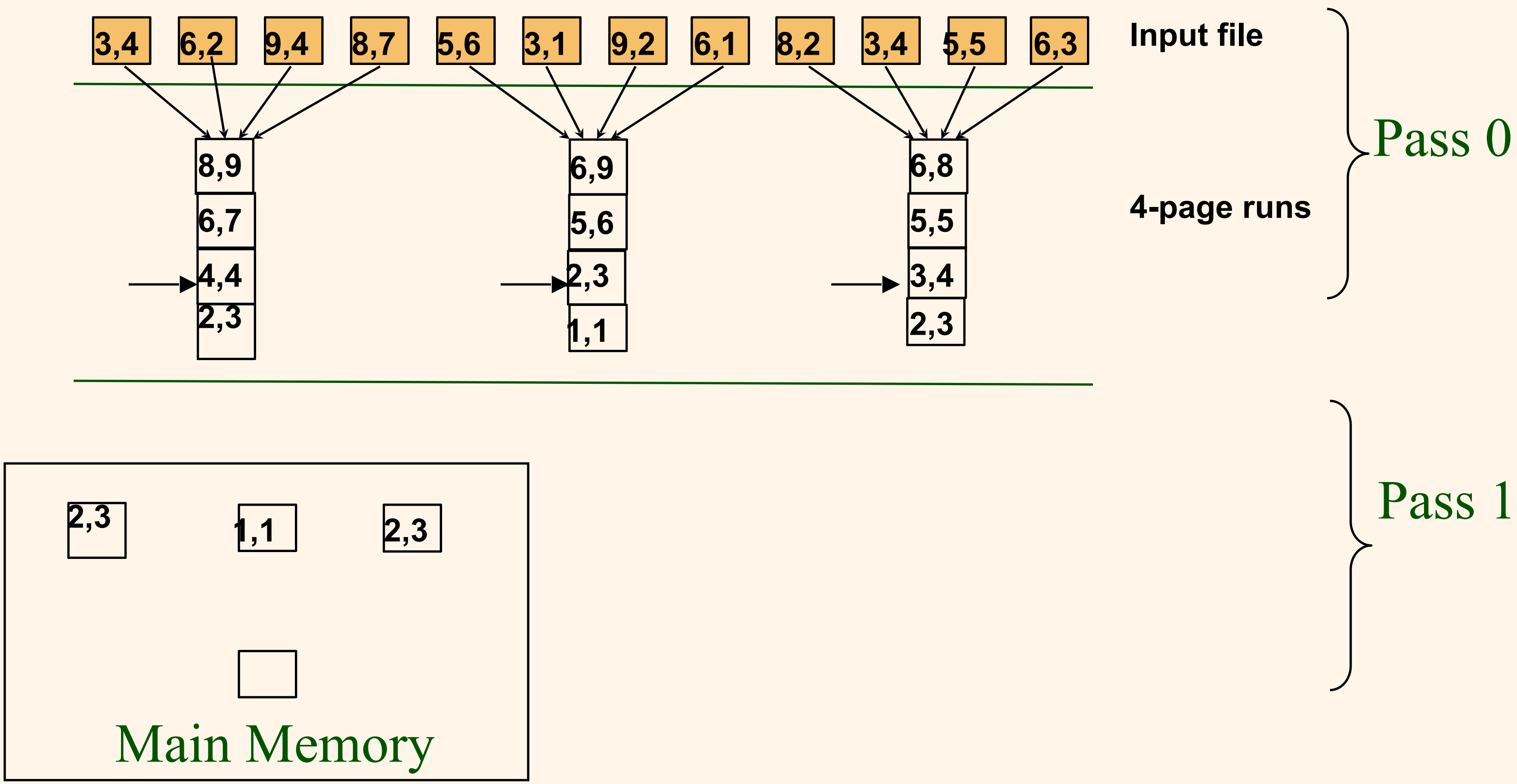
General External Merge Sort: Phase 2

- # buffer pages B = 4



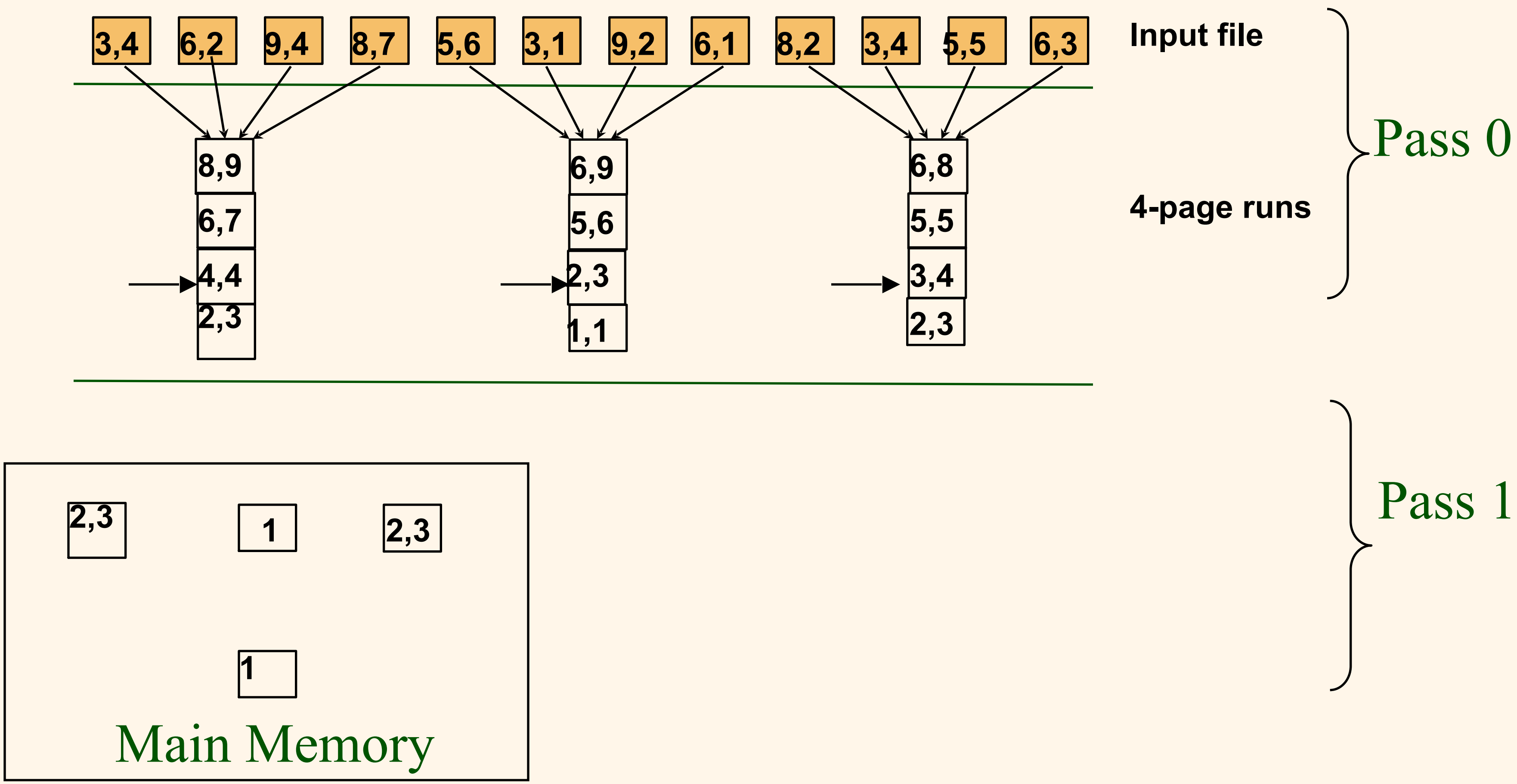
General External Merge Sort: Phase 2

- # buffer pages B = 4



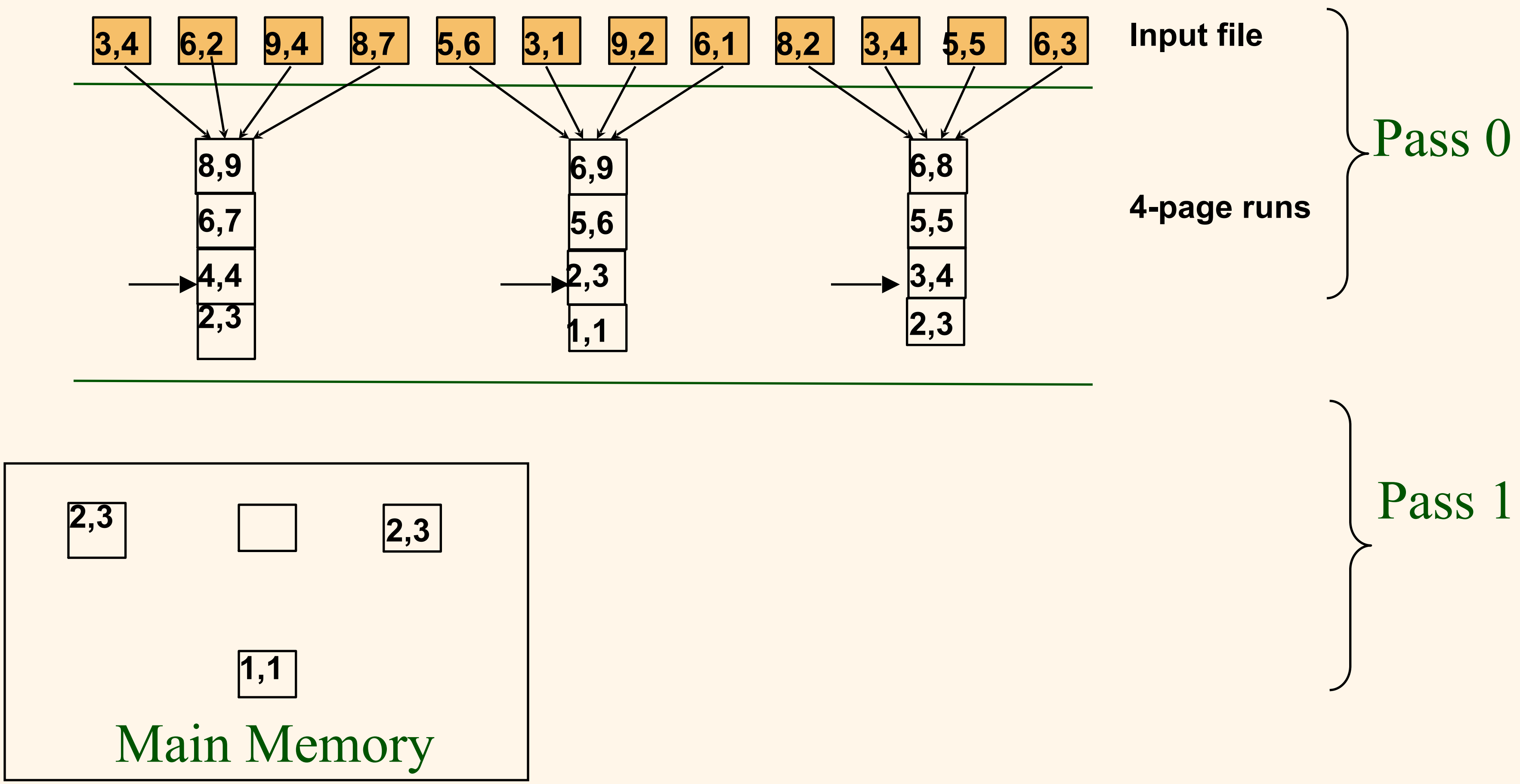
General External Merge Sort: Phase 2

- # buffer pages B = 4



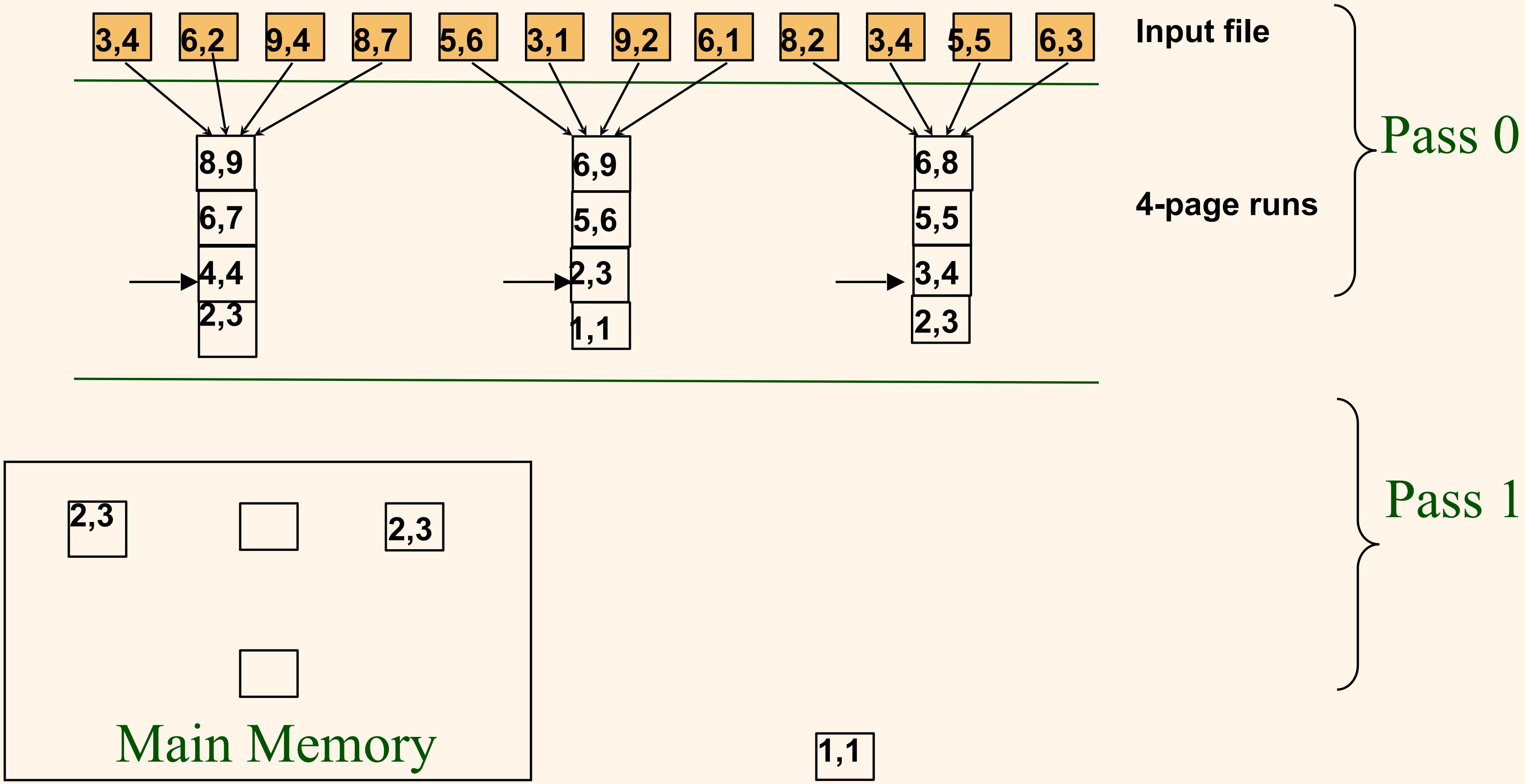
General External Merge Sort: Phase 2

- # buffer pages B = 4



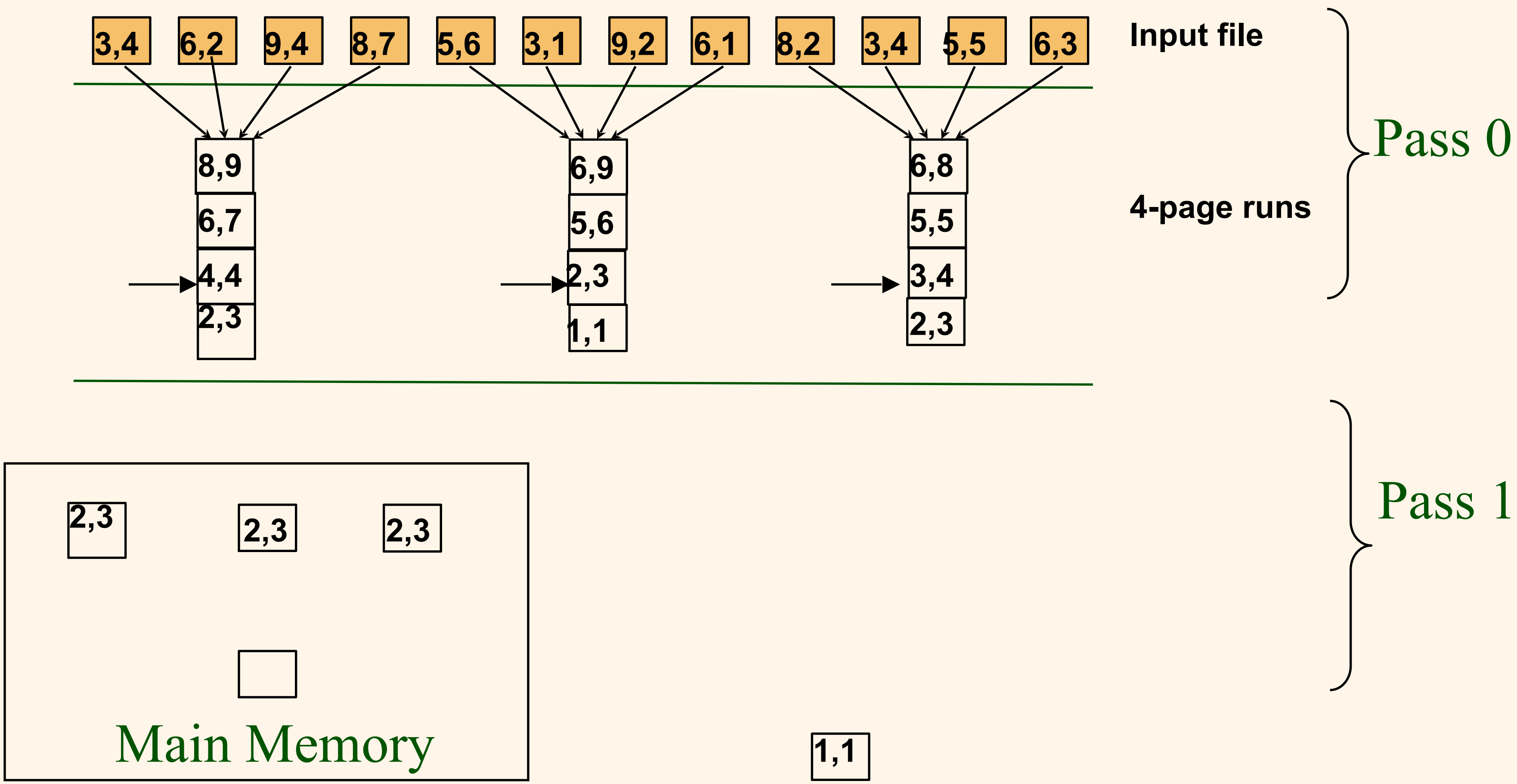
General External Merge Sort: Phase 2

- # buffer pages B = 4



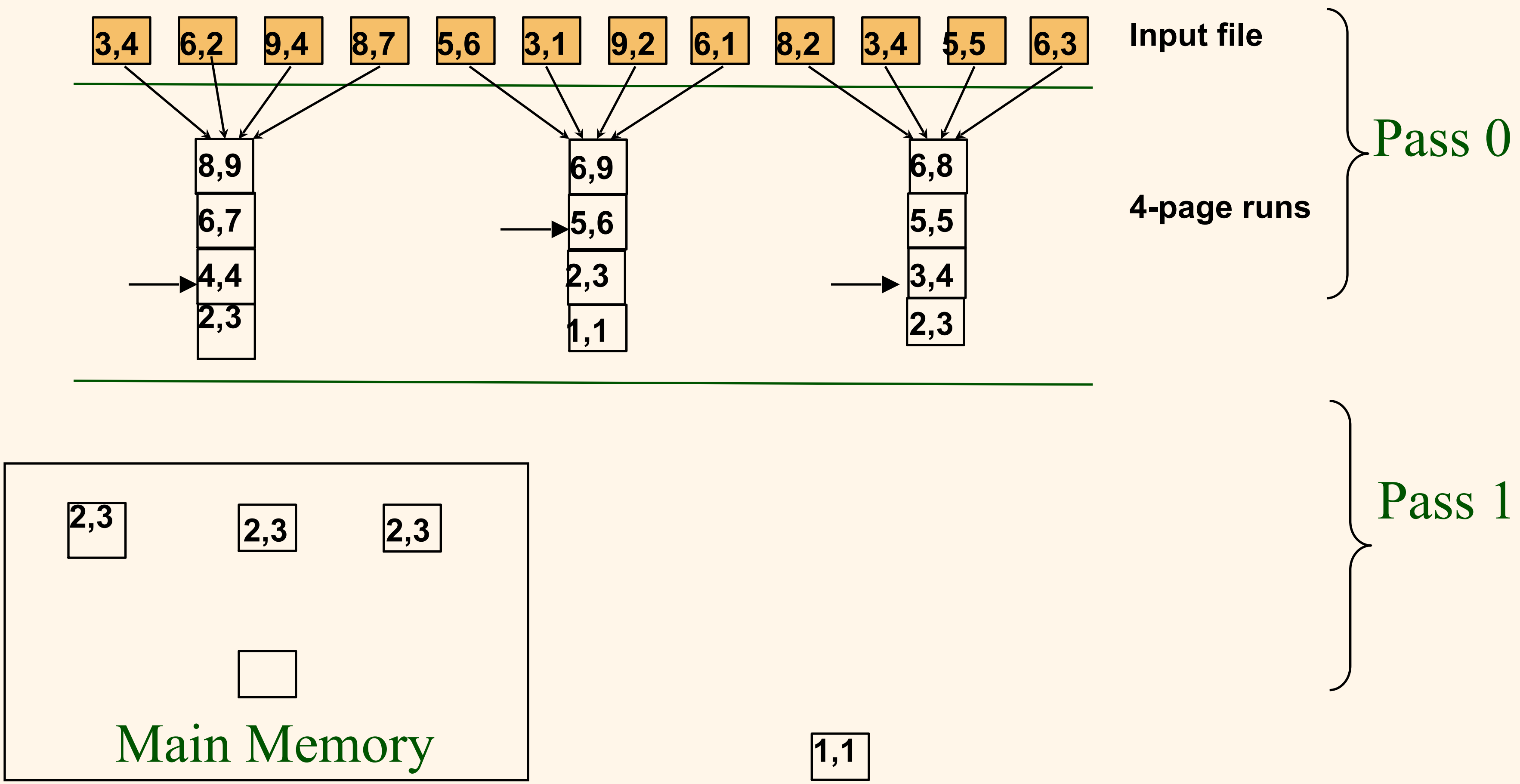
General External Merge Sort: Phase 2

- # buffer pages B = 4



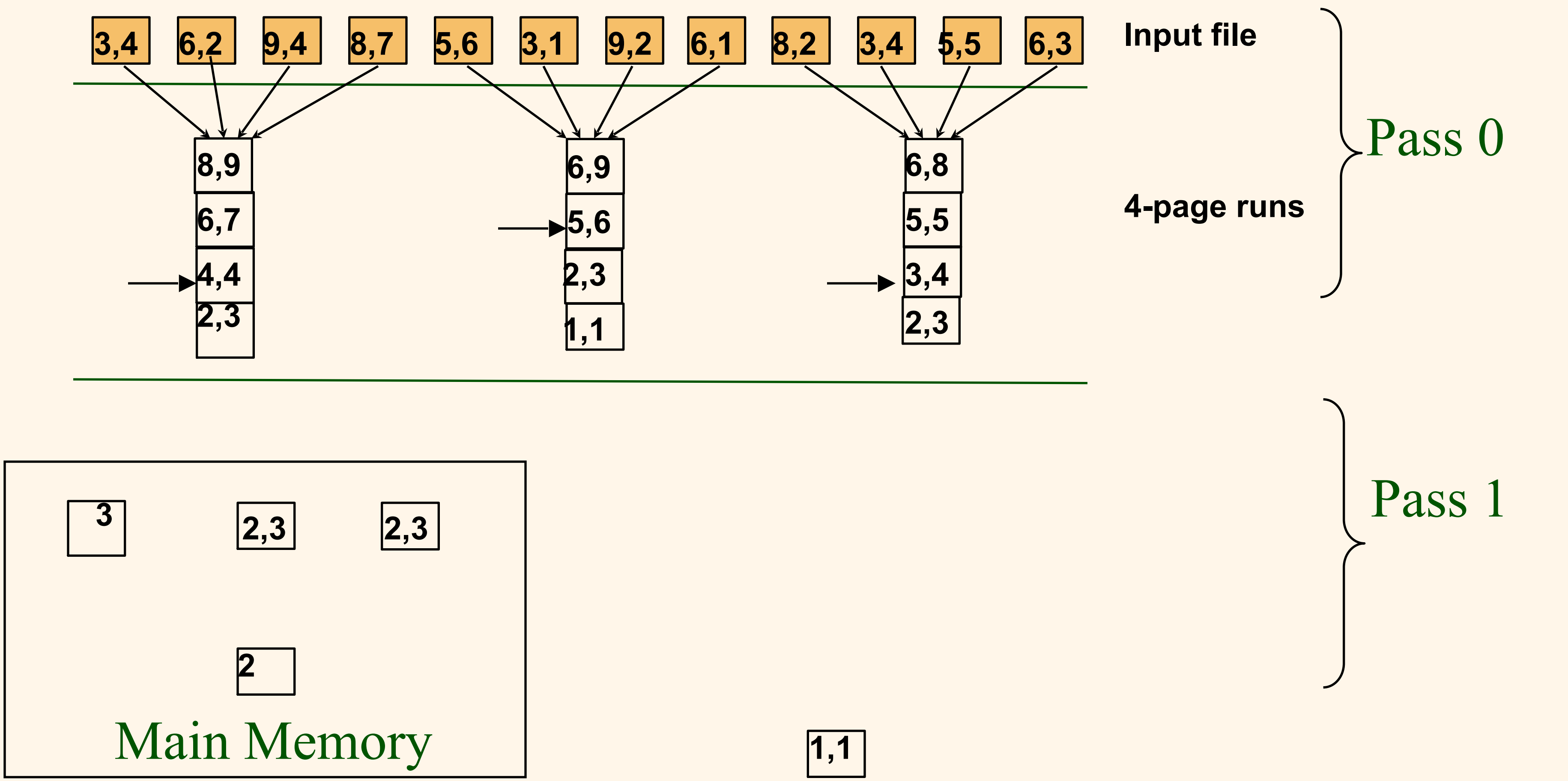
General External Merge Sort: Phase 2

- # buffer pages B = 4



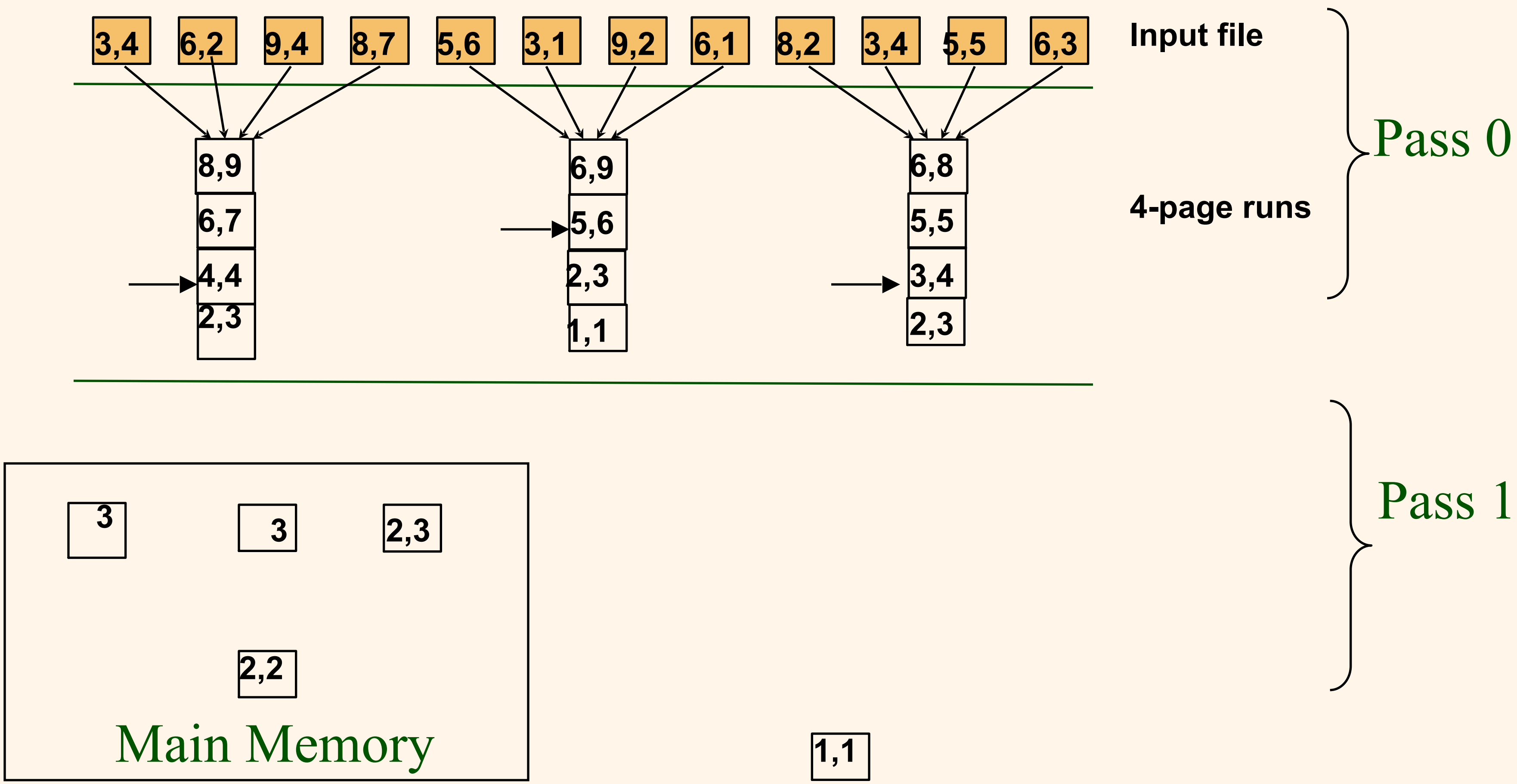
General External Merge Sort: Phase 2

- # buffer pages B = 4



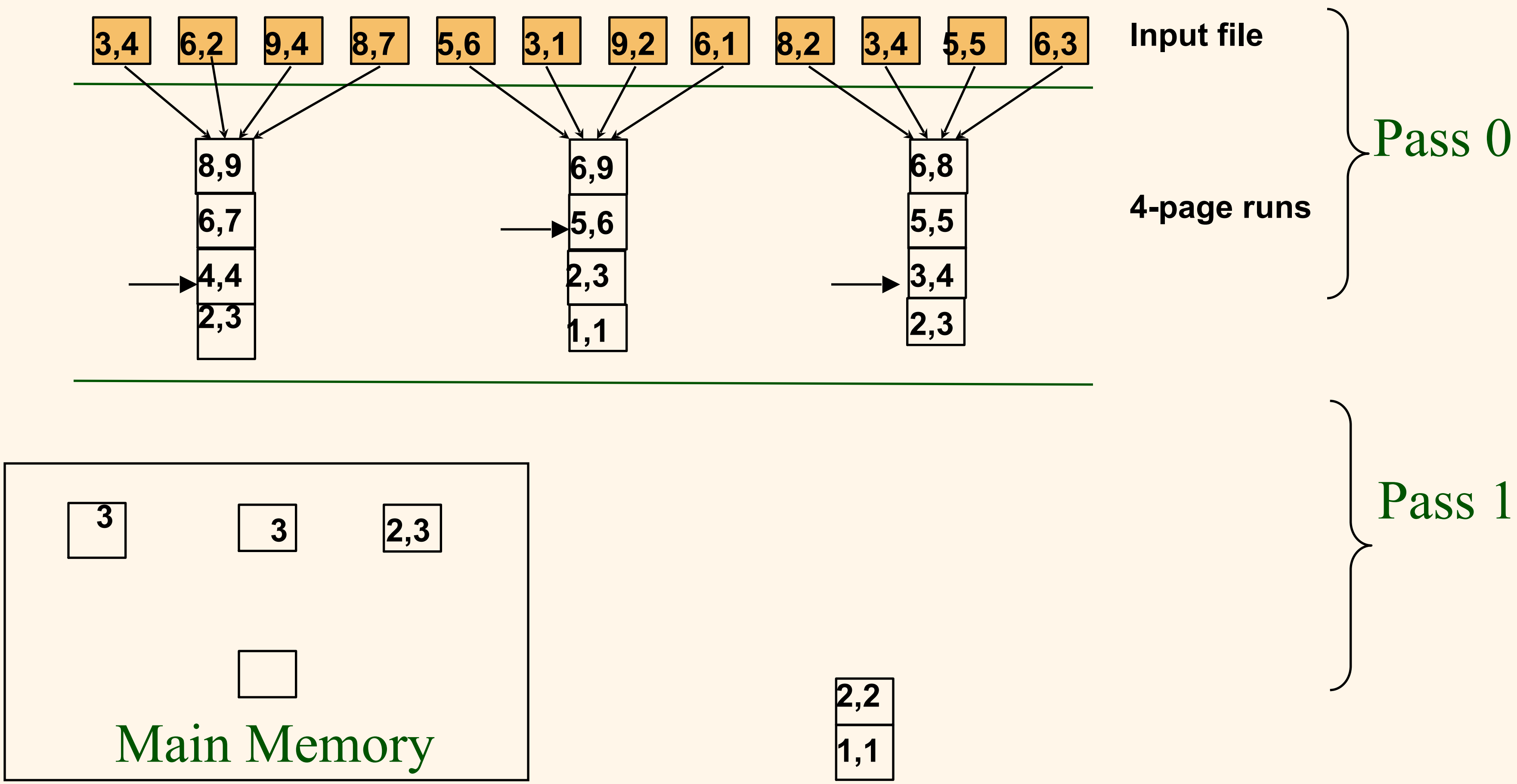
General External Merge Sort: Phase 2

- # buffer pages B = 4



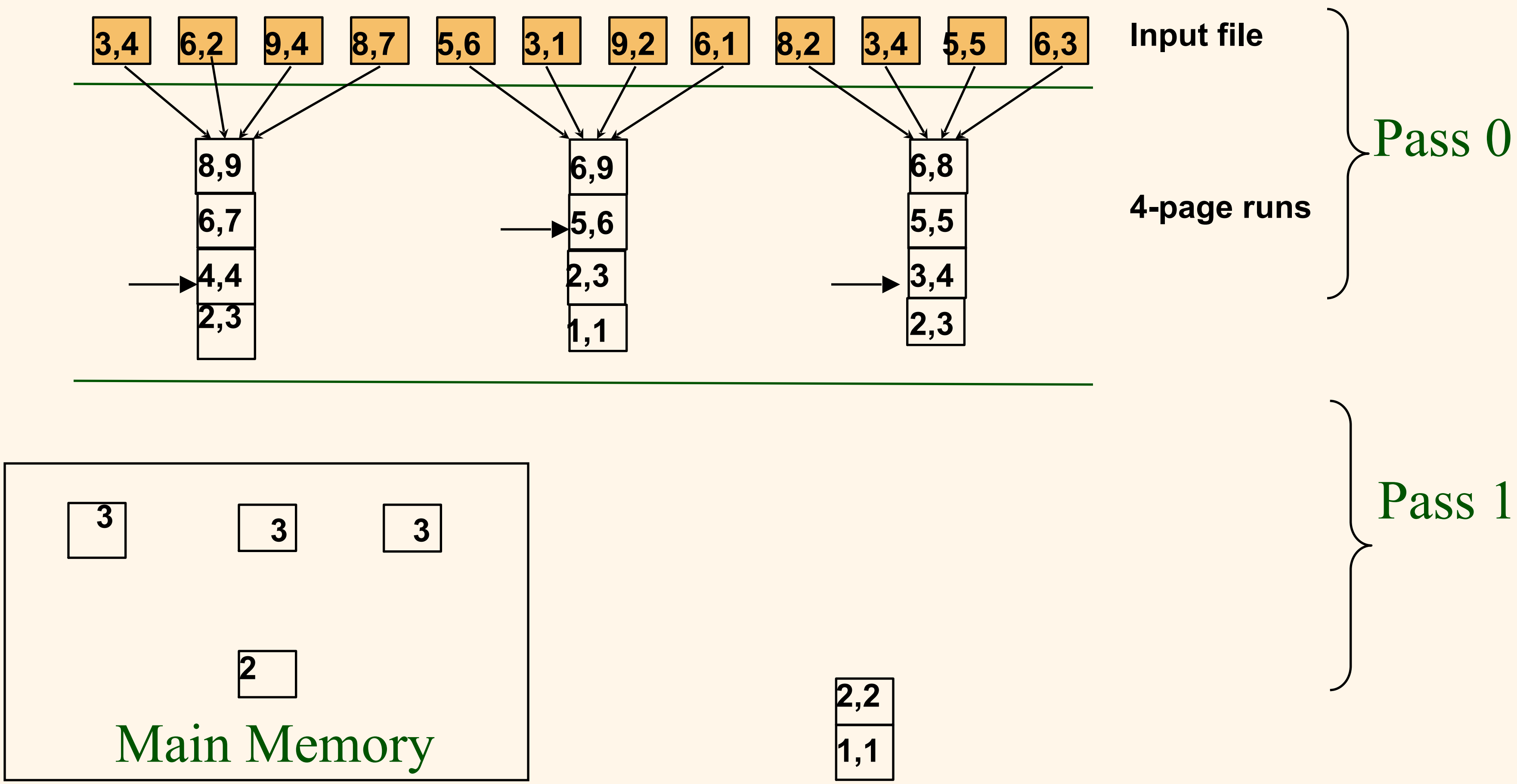
General External Merge Sort: Phase 2

- # buffer pages B = 4



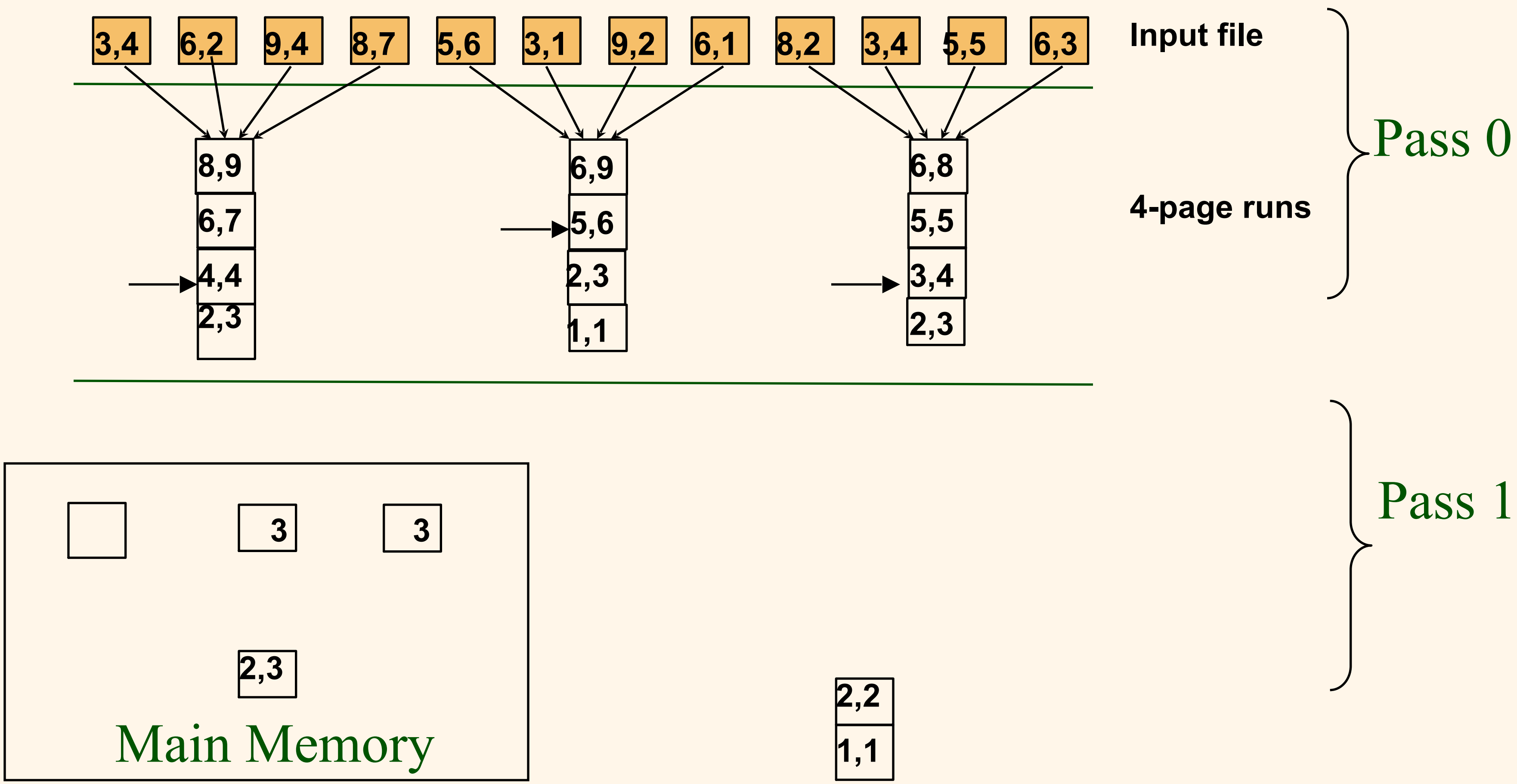
General External Merge Sort: Phase 2

- # buffer pages B = 4



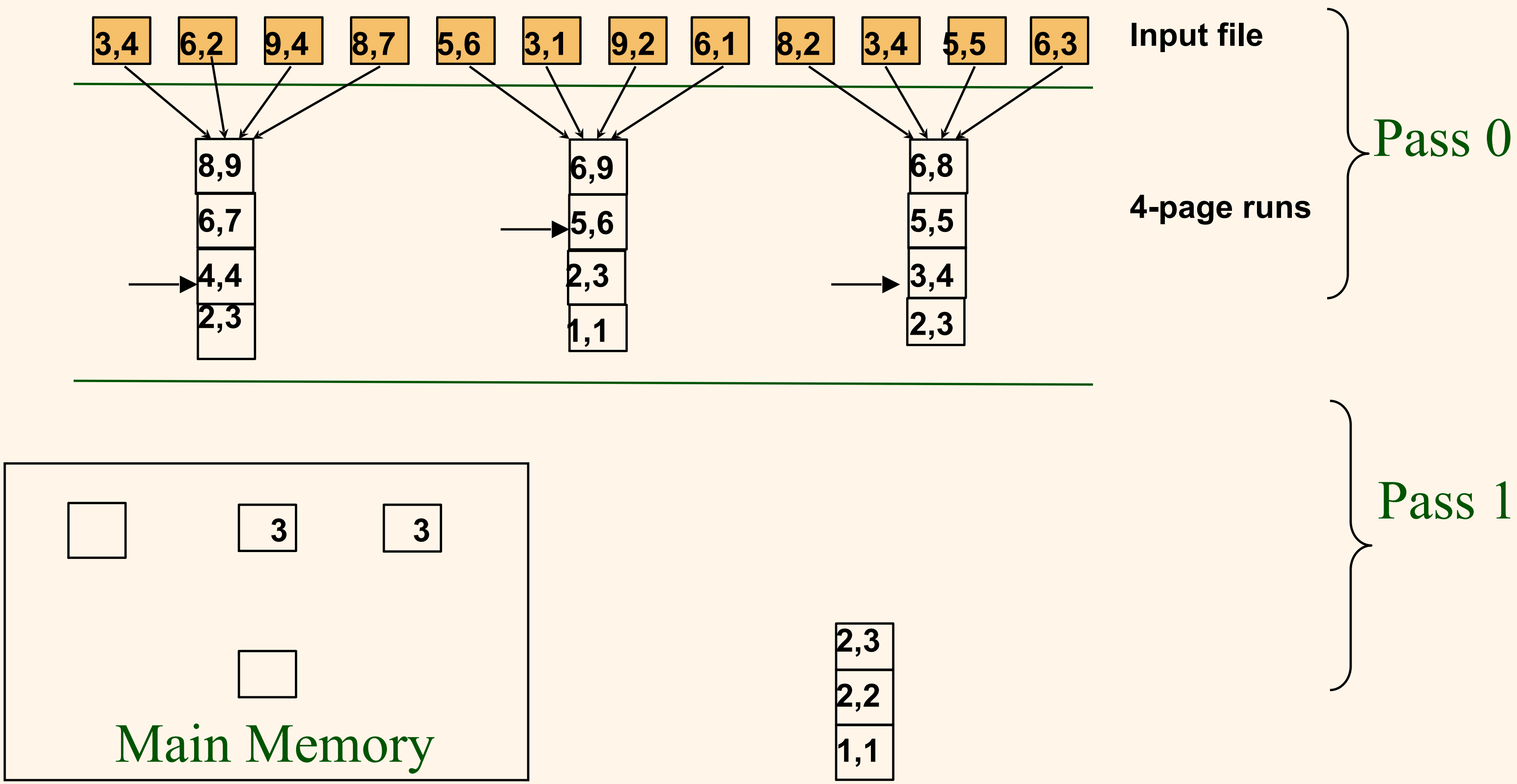
General External Merge Sort: Phase 2

- # buffer pages B = 4



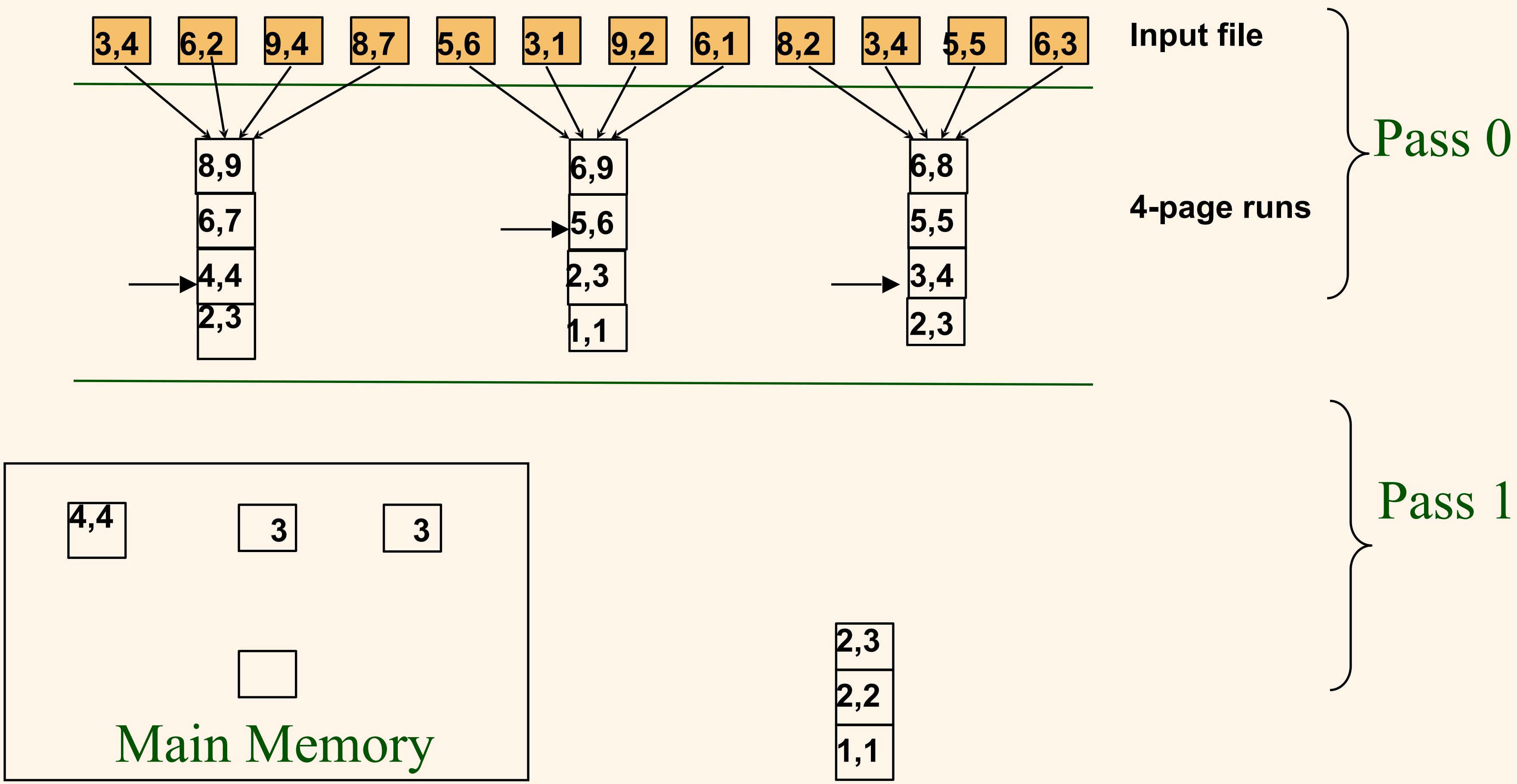
General External Merge Sort: Phase 2

- # buffer pages B = 4



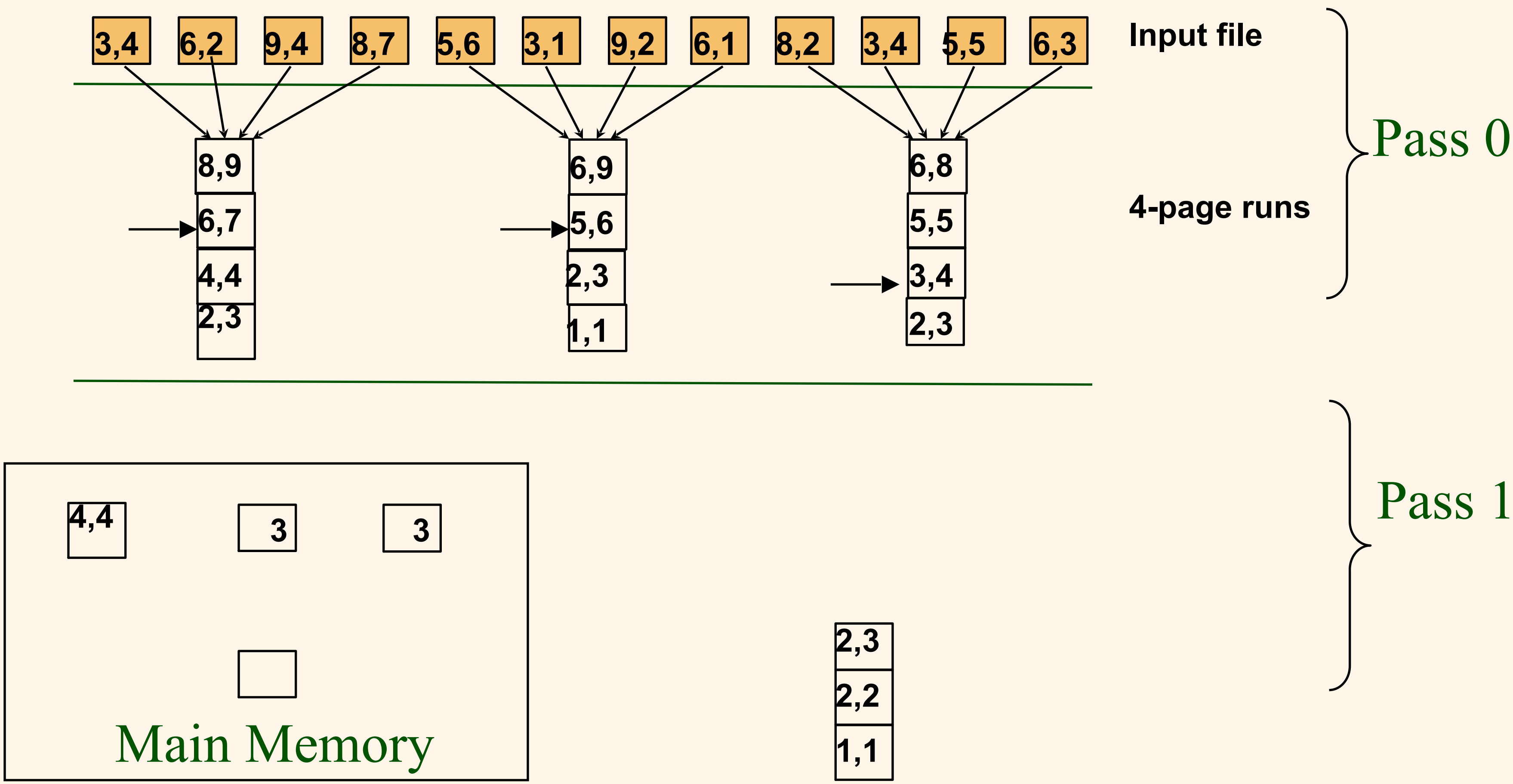
General External Merge Sort: Phase 2

- # buffer pages B = 4



General External Merge Sort: Phase 2

- # buffer pages B = 4



General External Merge Sort: Analysis

- Total I/O cost for sorting file with N pages
- Cost of Pass 0 = $2N$

- If # merge passes is P then: $B(B-1)^P = N$
- Therefore P = $\lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Cost of each merge pass = $2N$
- Cost of all merge passes = $2N \times \lceil \log_{B-1} \lceil N/B \rceil \rceil$

- Total cost = $2N\left(\lceil \log_{B-1} \lceil N/B \rceil \rceil + 1\right)$

Number of Passes of External Sort

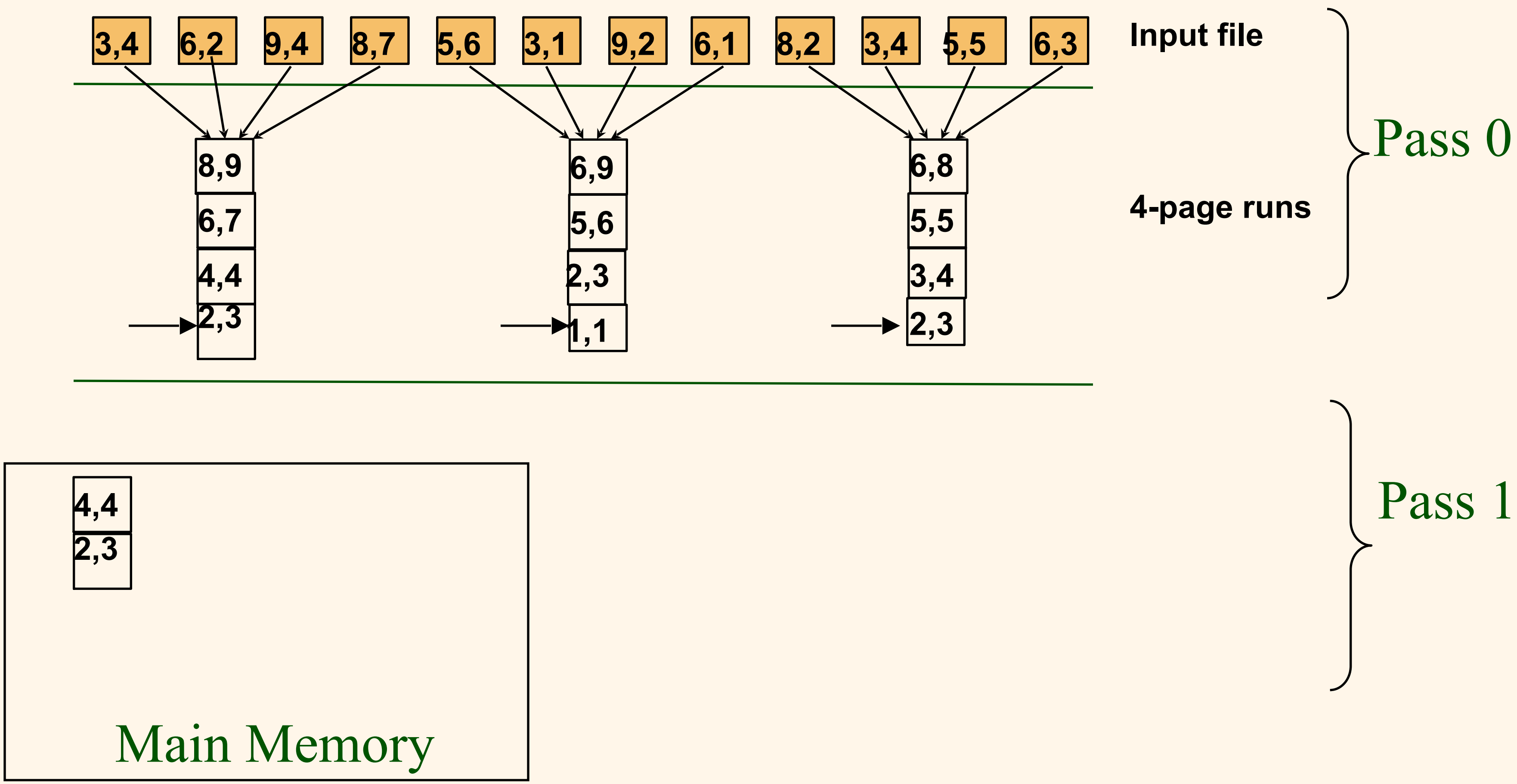
N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

External Merge Sort: Optimizations

- Currently, do one page I/O at a time
- But can read/write a *block* of pages sequentially!
  - Make each buffer input/output a block of pages

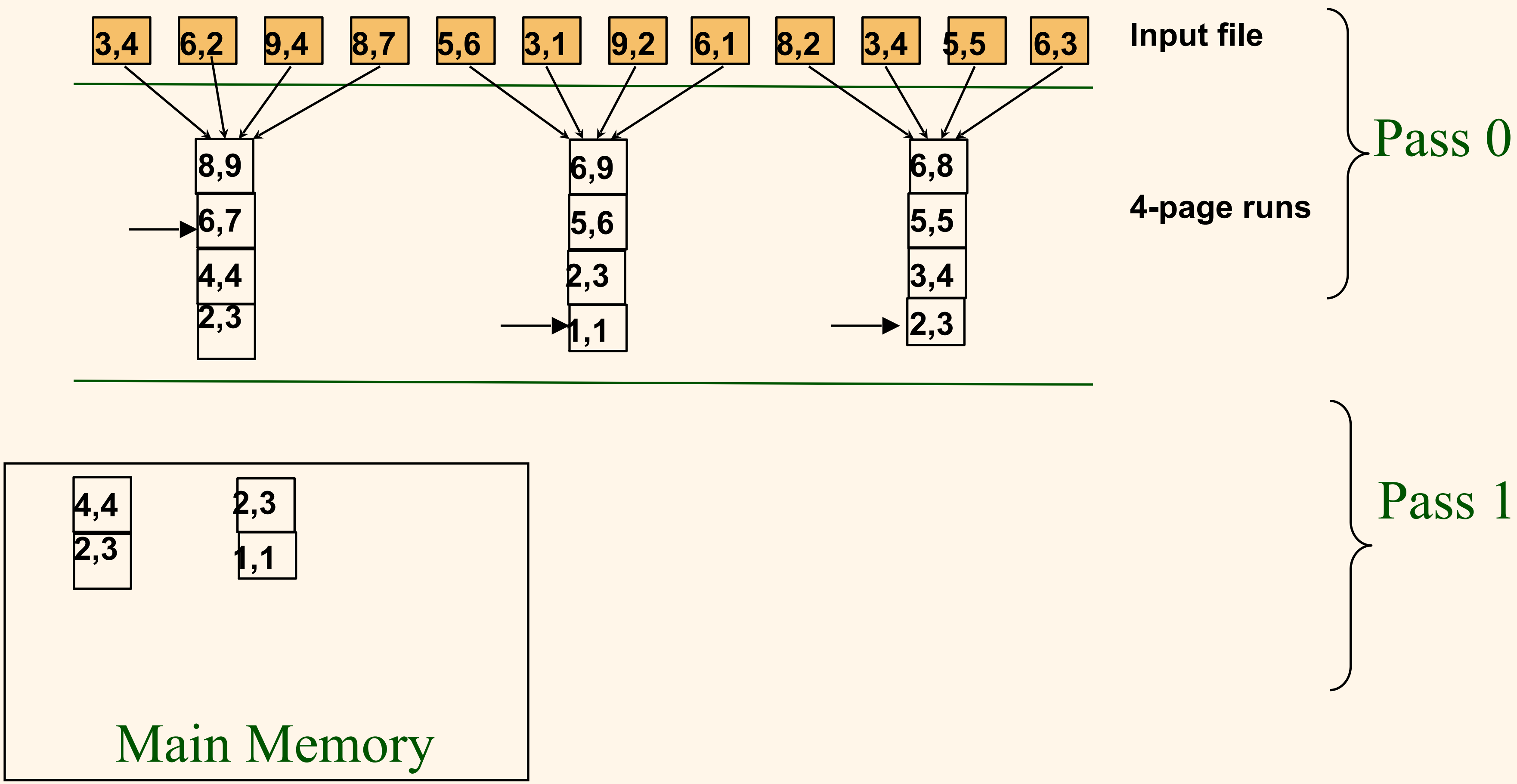
General External Merge Sort: Phase 2

- # buffer pages B = 8



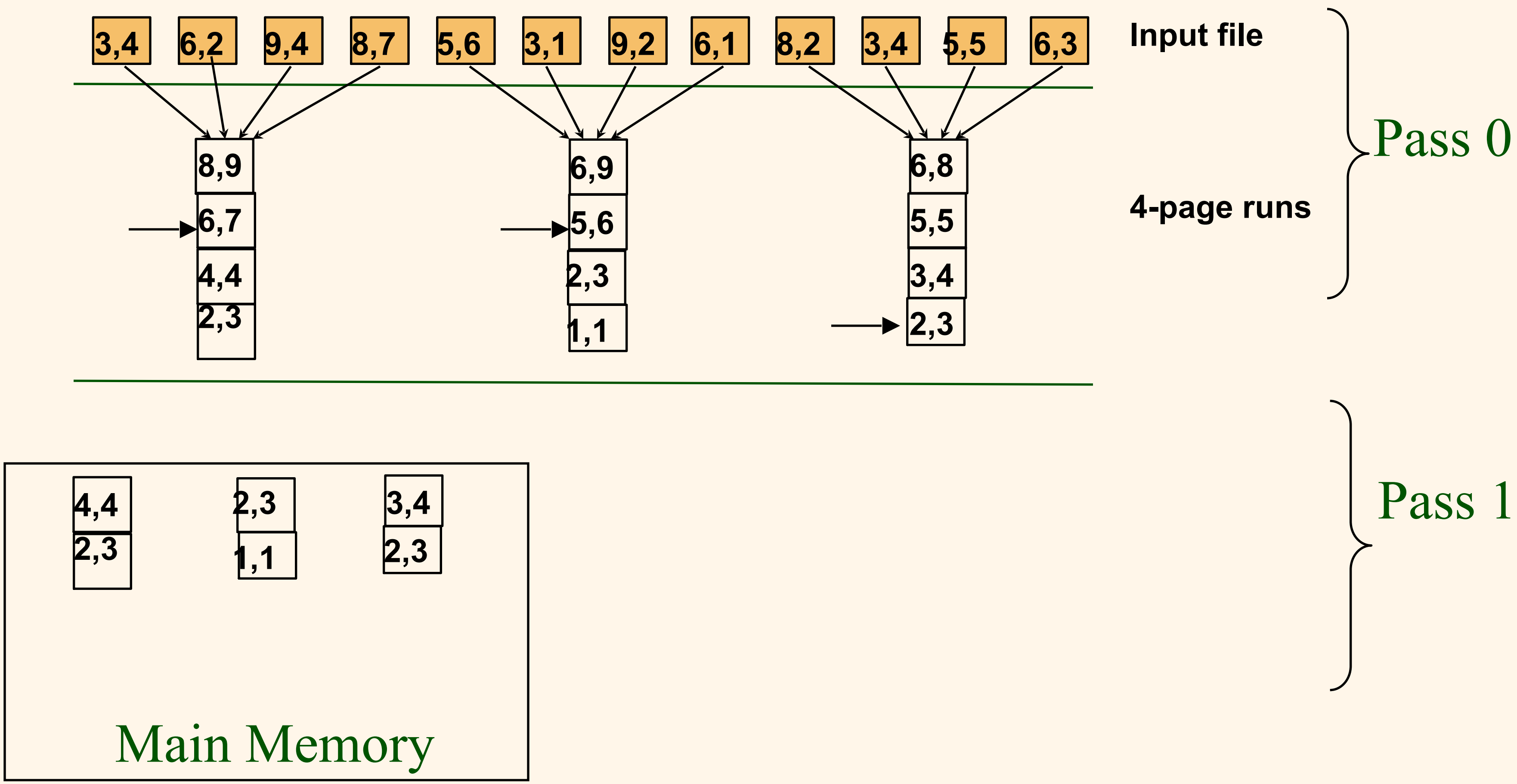
General External Merge Sort: Phase 2

- # buffer pages B = 8



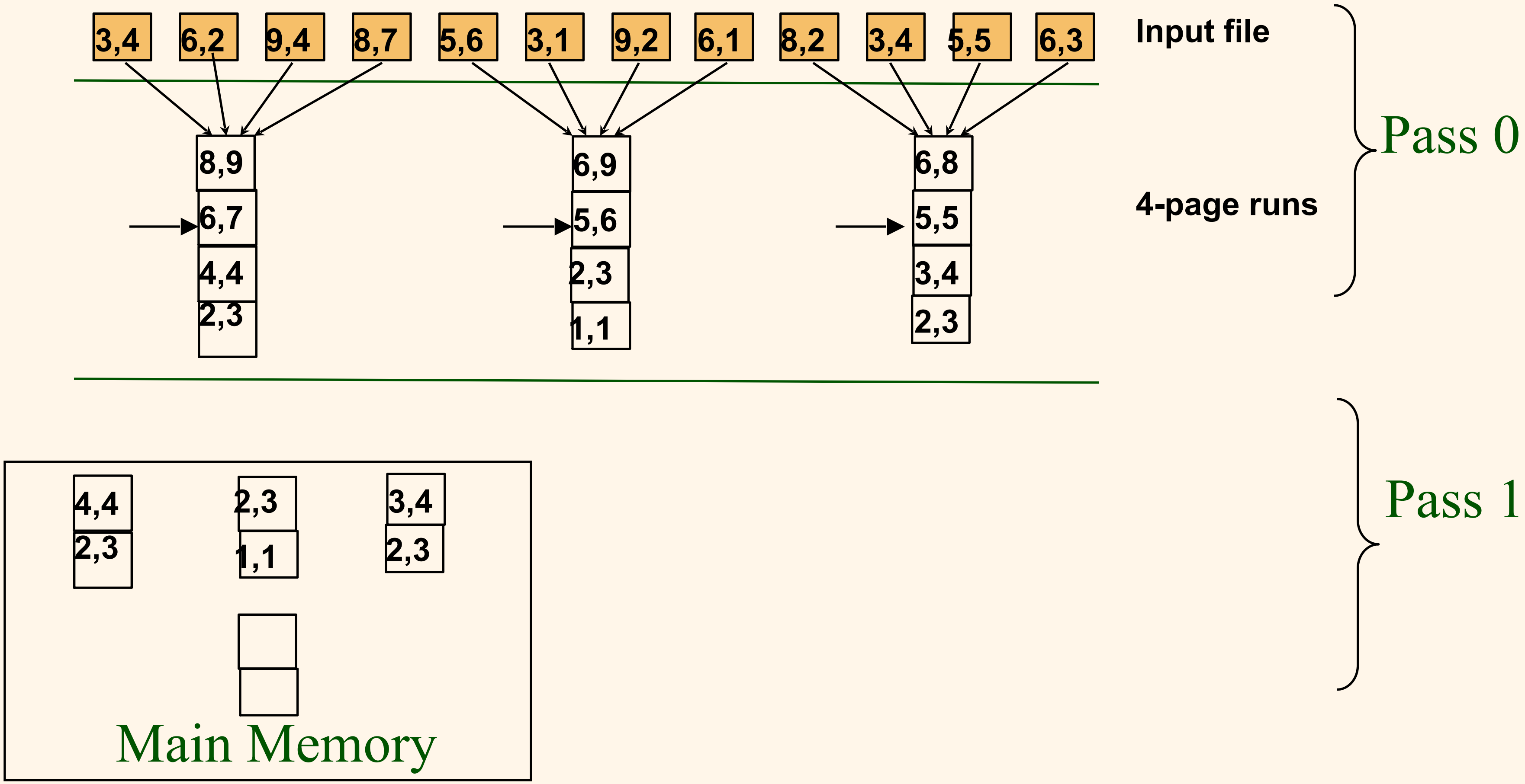
General External Merge Sort: Phase 2

- # buffer pages B = 8



General External Merge Sort: Phase 2

- # buffer pages B = 8

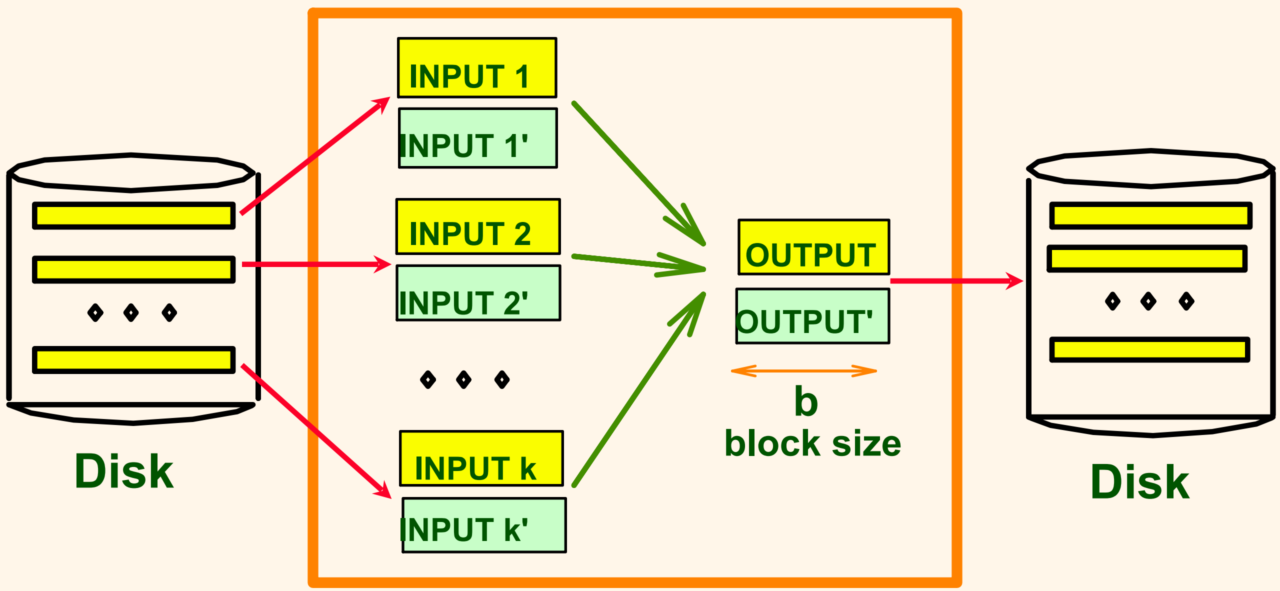


External Merge Sort: Optimizations

- Tradeoff: we reduce the merge fan-in
  - So more passes will be needed

General Merge Sort: Optimizations

- Double buffering: to reduce I/O wait time, *prefetch* into `shadow block'.
  - Again, potentially more passes; in practice, most files *still* sorted in 2-3 passes.



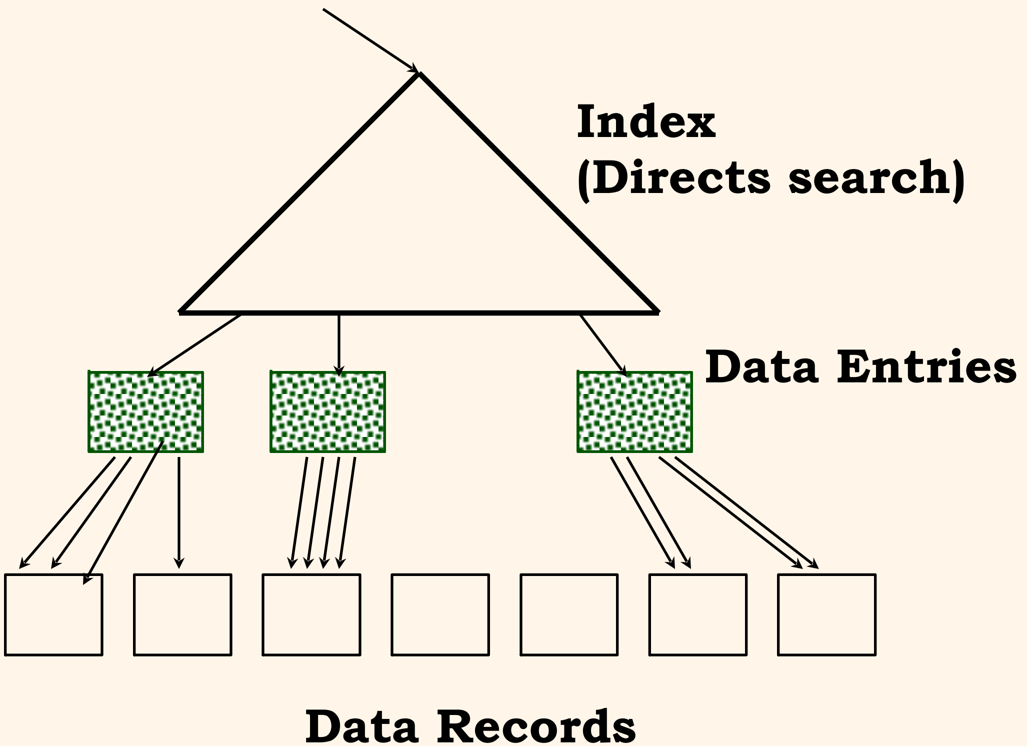
B main memory buffers, k-way merge

Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- ***Is this a good idea?***
- Cases to consider:
  - B+ tree is clustered — ***Good idea!***
  - B+ tree is not clustered — ***Could be a very bad idea!***

Clustered B+ Tree Used for Sorting

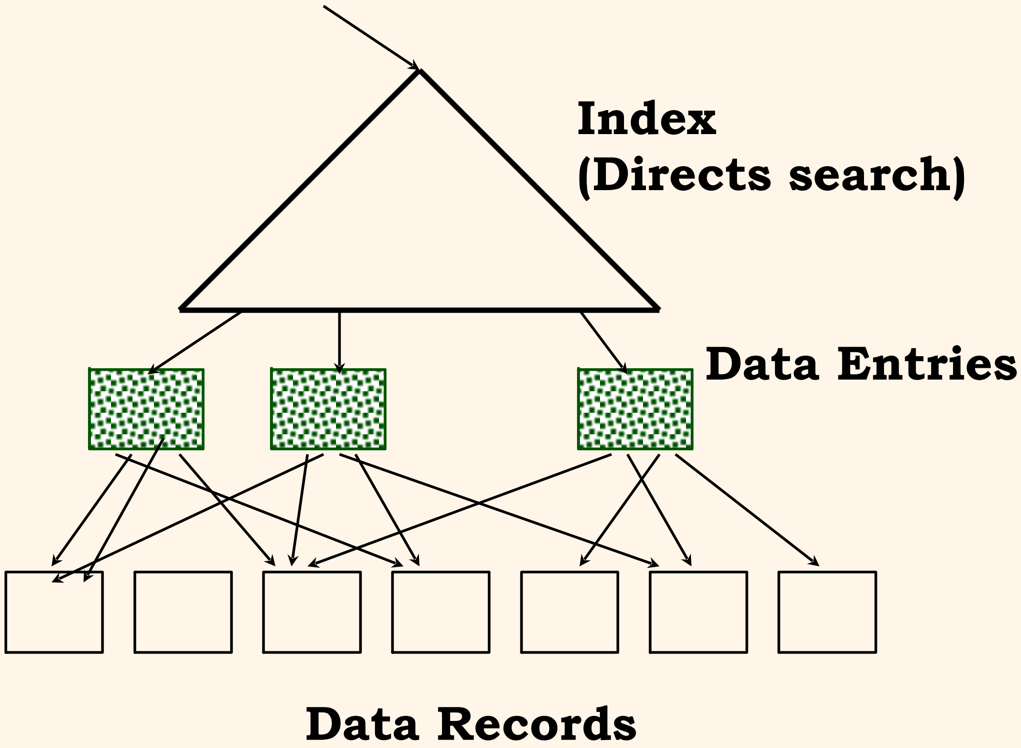
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.



- *Always better than external sorting!*

Unclustered B+ Tree Used for Sorting

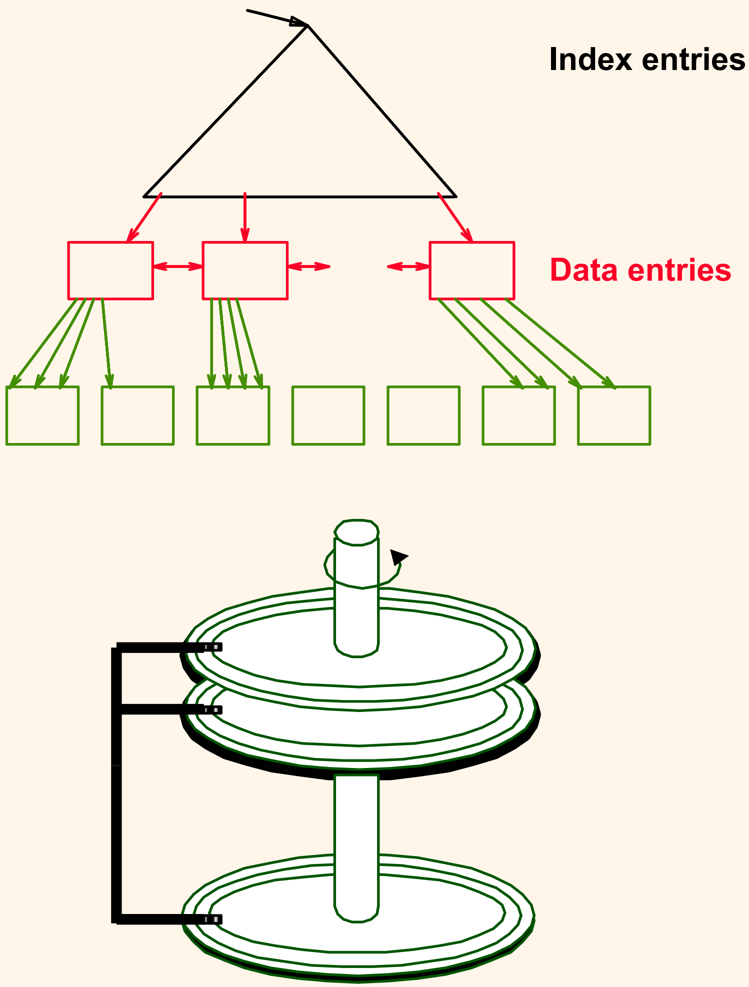
- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, one I/O per data record!



Summary

- External sorting is important
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages). Later passes: ***merge*** runs.
  - # of runs merged at a time depends on ***B***
  - In practice, # of passes rarely more than 2 or 3.

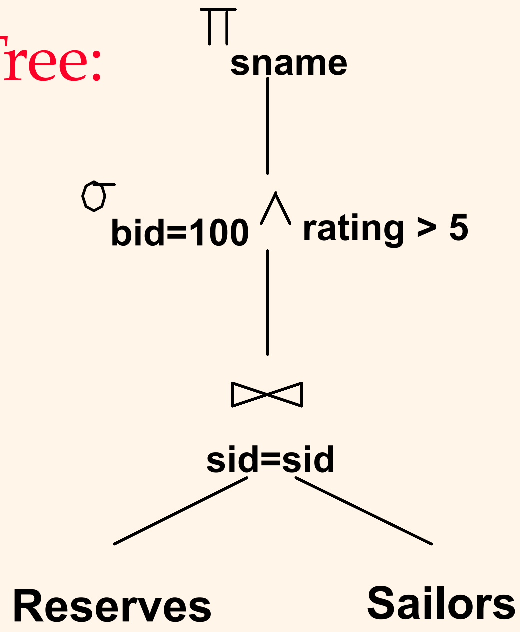
So far



So far

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

RA Tree:



Now for the "middle piece"

- How is a query actually evaluated in your DBMS?
- Two aspects:
  - How to implement each relational operator
    - Several options depending if indexes available
  - How to evaluate the whole plan
    - Maybe optimize plan first using RA equivalences
    - Choose implementation for each operator based on what's best for the overall plan (not just locally)

Implementing RA Operators

- Selection
- Projection
- Join
- Set operations
- Extended RA: GROUP BY, Aggregates

Readings

- [RG] Sec 14.1-14.3 (implementing select)

Example relations

- Sailors (sid, sname, rating, age)
- Reserves(sid, bid, day, rname)

- Each tuple of Sailors 50 bytes long (80 tuples per page) and have 500 pages
- Each tuple of Reserves 40 bytes long (100 tuples per page) and have 1000 pages

Cost model

- Number of page I/Os incurred by algorithm
- Ignore things like benefits of sequential access, CPU computation cost, etc.
- Crude but disk I/O is likely to dominate cost so a good way to compare implementation alternatives

Select Operator

```
SELECT  *
FROM    Reserves R
WHERE   R.rname = 'Alice';
```

Various cases

- Case 1: No index on any selection attribute
  - File could be sorted on selection attr. or not
- Case 2: Have “matching” index on all selection attributes
- Case 3: Have “matching” index on some (but not all) selection attributes
  - e.g. want to select "name = 'Alice' AND rating > 5" but only have index on name

No index on any selection attribute

- If file unsorted, need to scan whole thing
  - 1000 I/Os
  - Plus cost to write out results, but we ignore that aspect
    - Want to compare evaluation costs for different implementations
    - But all of them need to write out result
- If sorted, can do binary search
  - Logarithmic cost – for 1000 pages about 10 I/Os
  - Plus cost to retrieve qualifying tuples (0 to 1000 extra I/Os)

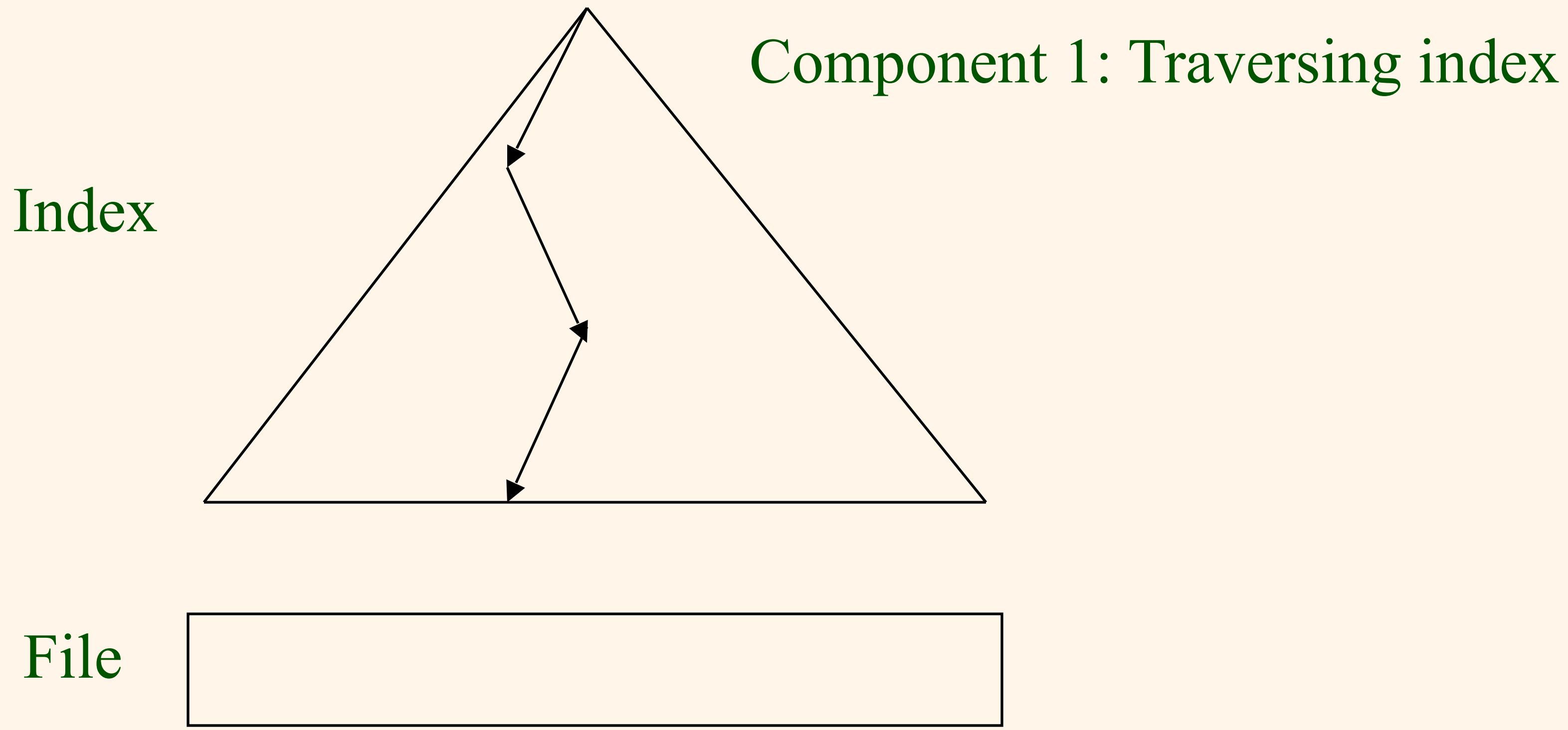
Using indexes for selection

- Start with simple case where only one selection attribute (e.g. "name = 'Alice'")
- Suppose we have a matching index
- Should we use it?
  - Probably, but not always!
  - Depends on cost of accessing data through index

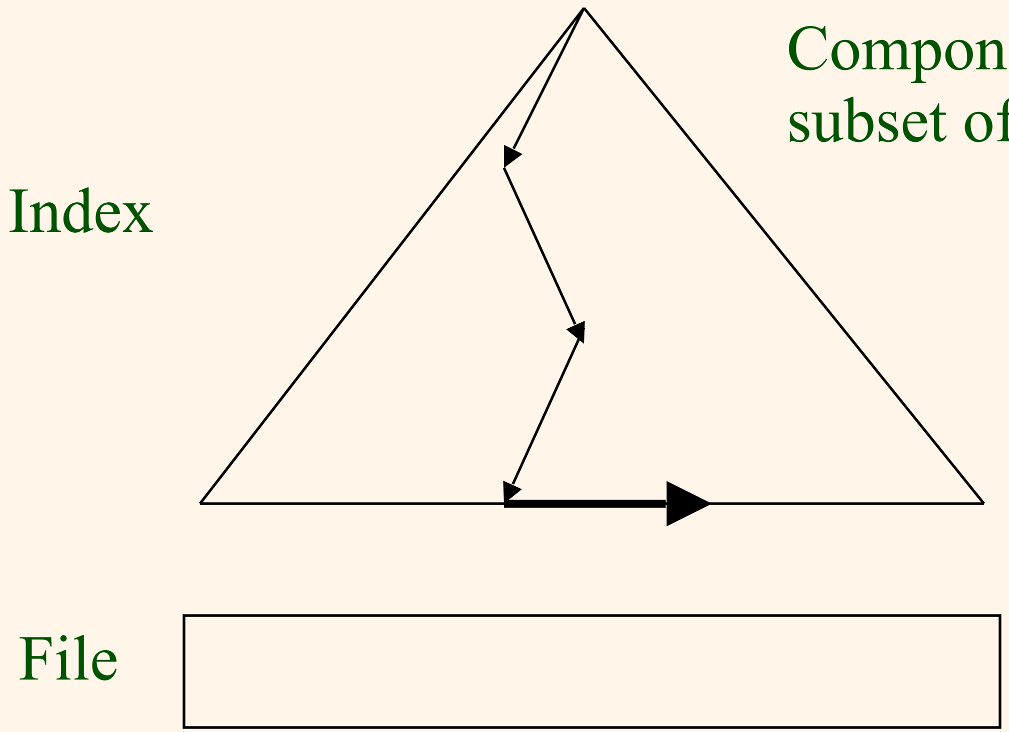
Using indexes for selection

- Tree index cost:
  - Traverse tree from root
    - Done only once
    - 2 to 3 I/Os (trees are short)
  - Scan leaf level pages to find all data entries
  - Retrieve actual data records

Cost Components – Tree Index

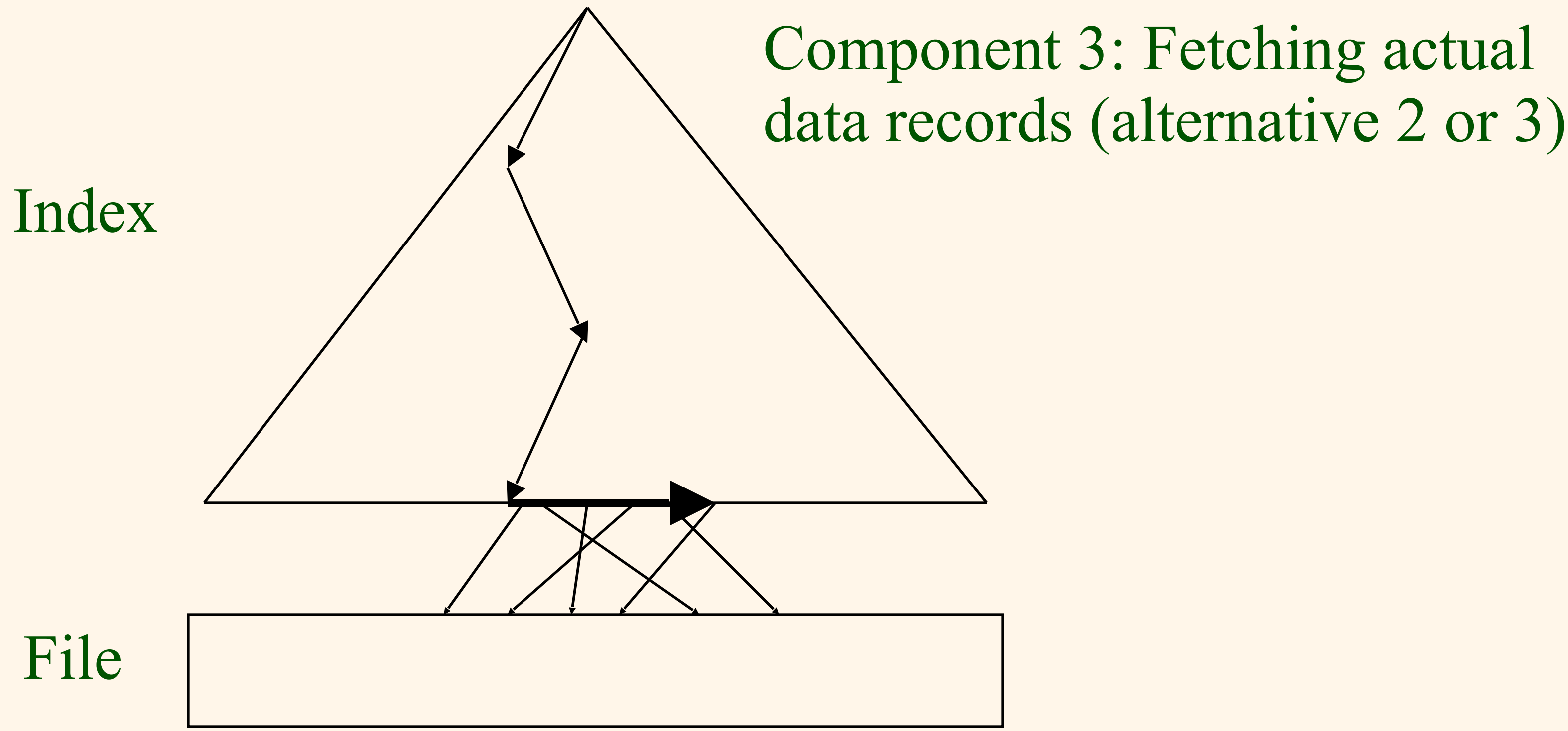


Cost Components – Tree Index



Component 2: Traversing subset of data entries in index

Cost Components – Tree Index



Clustered indexes

- If index is truly 100% clustered, don't need second component (scan from first leaf)
- But "clustered" may mean "data is stored close to search key order", not "exactly in search key order"
  - In this case do need to access leaf index pages
- In your calculations, both are ok but make it very clear which you're choosing and why
  - State your assumptions

What about hash indexes?

- Depends on implementation (linear, extendible etc)
- But typically small
- Reasonable assumption: 1 or 2 I/Os to find right bucket
  - Plus cost of retrieving actual data records (depends on number of matching records)

Example

- Selection on Reserves, condition is "R.rname < 'C%'"
- Assume uniform distribution of names, so about 10% of relation should be retrieved
  - 10K tuples, 100 pages

Example

- Have clustered B+ tree index on rname
  - Traverse index to find first leaf page - 1 or 2 I/Os
  - Start scan and retrieve tuples - 100 I/Os
- Have unclustered B+ tree index on rname
  - Worst case, retrieving each tuple requires a separate I/O so 10K I/Os
  - Much cheaper to just scan all 1000 original pages!
- Heuristic: scan cheaper than unclustered index if expect to retrieve more than 5% of tuples

Optimizations are possible

- Alternative 2 or 3, unclustered index
- Find qualifying data entries from index
- Sort the rids of the data entries to be retrieved
  - Remember rid = (page ID, slot #)
- Fetch rids in order
  - Ensures each data page is read from disk just once!
  - Although number of data pages retrieved still likely to be more than with clustering